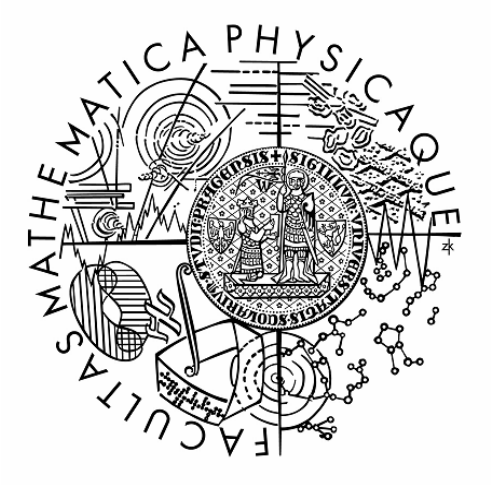
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Peter Piják

Interpret redukovaného Pascalu

Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: RNDr. Jiří Vyskočil
Studijní program: Informatika, Obecná informatika
2008

Poďakovanie

Ďakujem môjmu vedúcemu, RNDr. Jiřímu Vyskočilovi, za pomoc a pripomienky pri vývoji projektu a písaní práce.

Prehlásenie

Prehlasujem, že som túto bakalársku prácu napísal samostatne a výhradne s použitím uvedených prameňov. Súhlasím so zapožičaním práce.

V Prahe dňa 29. 5. 2008

Peter Piják

Obsah

Abstrakt

Názov práce: Interpret redukovaného Pascalu

Autor: Peter Piják

Katedra (ústav): Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: RNDr. Jiří Vyskočil

e-mail vedúceho: Jiri.Vyskocil@mff.cuni.cz

Abstrakt:

1. Nastudujte současné metody lexikální, syntaktické a sémantické analýzy zejména pro programovací jazyky (viz literatura).

2. Navrhněte redukci programovacího jazyka Pascal takovou, aby byla dobře implementovatelná (v podobě interpretu) a aby zachovala některé výhodné rysy programovacího jazyka Pascal.

3. V rámci návrhu redukovaného jazyka sestavte lexikální a syntaktická pravidla a specifikujte sémantiku takového jazyka.

4. Implementujte navržený programovací jazyk a odzkoušejte jeho správnou funkcionalitu na několika předem připravených testovacích příkladech.

Kľúčové slová: Pascal, prekladač, interpret, lexikálna a gramatická analýza

Title: An Interpreter of an Limited Pascal

Author: Peter Piják

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Vyskočil

Supervisor's e-mail address: Jiri.Vyskocil@mff.cuni.cz

Abstract:

1. Study contemporary methods for lexical, syntactic and semantic analysis for programming languages.

2. Define reduction of the programming language Pascal which may be implemented as an interpreter and which save some profitable Pascal's feature.

3. Define lexical and syntax rules and specify semantics of this language.

4. Implement the interpreter of defined language and test its functionality using some forward prepared examples.

Keywords: Pascal, compiler, interpreter, lexical and grammar analysis

1. Úvod

1.1. Motivácia

Práca programátorov je vysoko náchylná na vznik rôznych druhov chýb. Úplne zabránenie ich vzniku nie je možné. Kvôli ich odhaleniu sa v praxi na testovanie programov vynakladá vysoké množstvo času a finančných prostriedkov. Súčasné komerčné vývojové prostredia sa snažia preraziť najmä použitím prepracovaných ladiacich nástrojov.

Oblasťou, v ktorej najjednoduchšie a najčastejšie dochádza k vzniku programátorských chýb, je práca s ukazovateľmi a správou pamäte. Ich zvládnutie spôsobuje nemalé problémy programátorom začiatočníkom aj profesionálom. Používajú sa dve možnosti riešenia týchto chýb. Použitie špeciálnych knižníc na ladenie kódu alebo automatickej správy pamäte (garbage collector). Použitie metódy automatickej správy pamäte je jedným z dôvodov vysokého používania jazyka Java v súčasnosti.

1.2. Prínos a cieľ práce

1.2.1. Očakávaný prínos

Mojím cieľom je vytvoriť jednoduché vývojové prostredie, ktoré bude klásť vyšší dôraz na odhaľovanie chýb ako iné používané prostredia. Projekt pravdepodobne nebude konkurovať svojou rýchlosťou a nebude taký obsiahly, aby podporoval všetky techniky a knižnice zvoleného programovacieho jazyka. Bude podporovať iba definovanú redukciu programovacieho jazyka.

Aj preto nie je možné očakávať rozsiahle používanie projektu. Predpokladám, že po projekte môže siahnuť predovšetkým programátor študent pri vývoji menších programov práve kvôli lepšiemu zameraniu na odhaľovanie chýb.

1.2.2. Výber programovacieho jazyka

Dôležitou otázkou je výber programovacieho jazyka. Pri výbere treba zohľadniť orientáciu projektu pre programátorov začiatočníkov. Programátor začiatočník by si mal svoju chybu všimnúť, odstrániť a poučiť sa z nej. Jednoduchá chyba, ktorá vznikne ako preklep na klávesnici, by mala byť odhalená napríklad ako gramatická chyba. V tomto smere nie sú vyhovujúce jazyky C/C++.

V súčasnosti (rok 2008) je stále najfrekventovanejším programovacím jazykom používaným pri výučbe jazyk Pascal. Aj napriek tomu, že vznikol v roku 1970 [1] a v praxi sa už takmer nepoužíva. Obsahuje základné dátové štruktúry a techniky, ktoré by mal začínajúci programátor poznať. Prechod k iným, dnes používaných jazykom nebýva pre študentov výrazný problém. Preto som sa rozhodol pre výber jazyka Pascal.

1.2.3. Cieľ práce

Cieľom tejto práce je vytvoriť interpret programovacieho jazyka, ktorý vznikne redukciou jazyka Pascal. Úlohou tohto interpreta je vykonať príkazy vstupného kódu a odhaliť niektoré špecifické programátorské chyby. Projekt bude iba jednoduchá konzolová aplikácia, ktorá bude vykonávať vstupný zdrojový program a vypisovať informácie o chybách.

Je dôležitá presná formálna špecifikácia jazyka, ktorý bude interpret prijímať. Špecifikácia jazyka je náplňou kapitoly 2 a zaberá podstatnú časť tejto práce. Pre rýchly prehľad čitateľovi práce dobre poslúžia aj vygenerované syntaktické diagramy, ktoré sa nachádzajú v prílohe.

Podporovaný jazyk označím pracovným názvom Pascal light. Pri jeho definícii vychádzam z normy jazyka Pascal ISO/IEC 7185:1990 [2]. Nebudú podporované niektoré dátové štruktúry a konštrukcie, ktoré sú buď málo používané alebo je ich implementácia neprimerane zložitá. Ostanú zahrnuté vnorené funkcie špecifické pre Pascal, práca s ukazovateľmi a premennými umiestnenými na halde.

1.2.4. Odhaľované chyby

Odhaľované chýb počas interpretácie programu má byť prínosná vlastnosť projektu. Interpret musí kontrolovať lexikálnu, syntaktickú a sémantickú správnosť vstupného kódu. Pri interpretácií bude projekt kontrolovať, či jednotlivé príkazy môžu byť vykonané. Ak by nasledujúci príkaz spôsobil chybu, interpret musí podať informáciu o možnej chybe a zastaviť sa. Nie je prípustné, aby došlo k chybe interpreta a programátor sa nedozvedel, k akej chybe došlo.

Presný popis a spôsob odhaľovania chýb je bližšie popísaný v kapitole 3.

Interpret bude preverovať nasledujúce možné základné chyby:

- čítanie hodnoty neinicializovanej premennej
- delenie nulou a zvyšok pri delení nulou

- pretečenie a podtečenie hodnoty základných typov a reťazcov
- prístup mimo hranice poľa

Najviac chýb nastáva pri práci s ukazovateľmi. Projekt nebude implementovať v Pascale automatickú správu pamäte. Tú ponechávam na programátorovi. Interpret bude iba kontroluje správnosť práce s ukazovateľmi a dostupnosť pamäte na halde nejakým ukazovateľom.

Pri práci s ukazovateľmi preverí vznik nasledujúcich chýb:

- čítanie hodnoty ukazovateľa, ktorý nebol inicializovaný
- dereferencia ukazovateľa (operátorom `^`) s hodnotou `Nil`
- dereferencia ukazovateľa, ktorý referuje na premennú, ktorá bola medzitým uvoľnená zo zásobníka alebo z haldy
- použitie operátora `dispose` na premennú, ktorá je umiestnená na zásobníku
- použitie operátora `dispose` na premennú, ktorá je iba časťou alokovanej štruktúrovanej premennej záznam alebo pole
- bol odstránený jediný ukazovateľ na pamäť na halde, ostala nedostupná alokovaná pamäť

1.3. Alternatívny projekt - IPcute

IPcute [3] je vývojové prostredie od Françoisa Breta z Univerzity v Tours. Projekt je interpret programov v jazyku Pascal a C. Podľa názvu (cute), prostredie sa snaží byť milé a názorné. Program umožňuje všetko, čo zvládajú známe vývojové prostredia. Zobrazuje hodnoty premenných a zásobník volania funkcií. Zaujímavé je grafické zobrazovanie ukazovateľov premenných umiestnených na halde. Nevýhodou je neintuitívnosť používateľského rozhrania a vysoká nestabilita programu.

Interpret, ktorý je súčasťou tejto bakalárskej práce, sa oproti projektu IPcute nezameria na používateľské rozhranie, ale na stabilitu a odolnosť.

2. Špecifikácia jazyka Pascal light

2.1. Charakteristika

Je potrebné presne špecifikovať redukciu jazyka Pascal, ktorú bude interpret rozpoznávať a interpretovať. Pascal bol prvýkrát definovaný [1] v roku 1983 medzinárodným štandardom ISO/IEC 7185. Cieľom bolo upraviť pôvodný Wirthov jazyk. V roku 1989 dochádza k revízii štandardu a vzniká norma ISO 7185:1990 [2]. Príčinou je existencia chýb a početných viaczmyselných definícií. Jazyk Pascal light špecifikovaný v mojej práci vychádza z tohto dokumentu.

Štandardizovaný Pascal zredukujem o niektoré dátové štruktúry a konštrukcie, ktoré sú buď málo používané alebo ich implementácia nie je vhodná pre interpret. Medzi špecifické vlastnosti jazyka Pascal, ktoré budú podporované, patria: vnorené funkcie, procedúry pre vstup a výstup (`writeln`, `readln`) s premenlivým počtom parametrov, polia s možnosťou definície hraničných indexov. Kvôli zložitej implementácii, prílišnej špecifickosti alebo nízkej použiteľnosti nie sú podporované napríklad typy: množina, interval a vymenovaný typ (enum), vstup a výstup do súborov, návesti (`label`), skoky (`goto`) a vetvenie (`case`). Do jazyka pridám niektoré užitočné riešenia, ktoré použila firma Borland v prostredí Turbo Pascal [4].

2.2. Konvencie pri definícii jazyka

Úlohou tejto kapitoly je definovať lexikálne, syntaktické pravidlá a sémantiku podporovaných konštrukcií. V tejto sekcii uvediem použité konvencie pri ich definícii. Pravidlá budú zapísané pomocou rozšírenej Backusovej-Naurovej formy.

Význam použitých symbolov:

<Arial>	neterminálny symbol
`Courier New`	terminálny symbol
::=	definícia prepisovacieho pravidla
|	alternatíva
.	koniec definície
[x]	0 alebo 1 inštancia x
{x}	0 alebo viac inštancií x

2.3. Lexikálne pravidlá

2.3.1. Všeobecné pravidlá

V Lexikálnych pravidlách nie sú výrazné zmeny oproti norme Pascalu.

<Písmeno>::= `A` | … | `Z` | a | … | z.

<Číslica>::= `0` | … | 9.

<Identifikátor>::= <Písmeno> { <Písmeno> | <Číslica> }.

Identifikátory začínajú písmenom. Ich dĺžka nie je obmedzená a na veľkosti pri porovnávaní znakov nezáleží.

2.3.2. Špeciálne symboly

<Špeciálny symbol>::=
<Slovný symbol> | <Operátor>.

<Slovný symbol>::=
`PROGRAM` | `CONST` | `TYPE` | `VAR` | `BEGIN` | `END` | `PROCEDURE` | `FUNCTION` | `FORWARD` | `ARRAY` | `OF` | `IF` | `THEN` | `ELSE` | `WHILE` | `DO` | `REPEAT` | `UNTIL` | `FOR` | `RECORD` | `TO` | `DOWNTO` | `READ` | `READLN` | `WRITE` | `WRITELN` | <Príkazový symbol> | `{(B|b)+|–}` .

Slovné symboly nie je možné použiť ako identifikátory.

<Príkazový symbol>::=
`BREAK` | `CONTINUE` | `EXIT` | `HALT`.

<Operátor>::=
<Unárny operátor> | <Binárny operátor> | <Operátor premennej>.

<Unárny operátor>::=
`NOT` | `ADDR` | `@` | `HIGH` | `LOW` | `ODD` | ABS | `PRED` | `SUCC` | `INT` | `FRAC` | `TRUNC` | `LENGTH`.

<Binárny operátor>::=
= | <= | >= | <> | < | > | + | – | * | / | `OR` | `XOR` | `DIV` | `MOD` | `AND`.

<Operátor premennej>::=
`DEC` | `INC` | `NEW` | `DISPOSE`.

2.3.3. Číselné a textové literálne konštanty

Je možné vytvoriť literálne konštanty celočíselného, reálneho a textového typu. Ich hodnoty musia byť v platnom rozsahu, číselné konštanty sú nezáporné. <Integer konštanta> je typu `Integer`, <Reálna konštanta> je typu `Real`. Ak je dĺžka neterminálu <Textová konštanta> jedna, tak je typu `Char`, inak typu `String`.

Vloženie znaku apostrof do textovej konštanty je možné sekvenciou dvoch apostrofov ako v Pascale.

<Integer konštanta>::=
    <Číslica> { <Číslica> }.
<Reálna konštanta>::=
    <Integer konštanta> . <Integer konštanta>
    [E|e [+|–] <Integer konštanta>] |
    <Integer konštanta> E|e [+|–] <Integer konštanta>.
<Textová konštanta>::=
    ‘ { <Ľubovoľný znak> } ‘.

2.3.4. Oddeľovacie znaky a komentáre

Oddeľovacími znakmi jednotlivých lexikálnych žetónov (tokens) sú: medzera, tabulátor, nový riadok, `;`, `..`, `,`, =, `:`, `:=`, `(`, `)`, `..`, `[`, `],` ^ a komentáre.

Sú definované dva druhy komentárov. Podľa oddeľovacích znakov, buď hranaté zátvorky `{` komentár `}`, alebo zátvorky s hviezdičkou `(*` komentár `*)`. Podľa štandardu Pascala je možné komentár začať znakom `{` a ukončiť `*)`, resp. obrátene. V tu definovanom jazyku sú tieto komentáre nezávislé.

2.4. Syntaktické a sémantické pravidlá - hlavička a definície

Program má podobne ako v Pascale povinnú hlavičku s identifikátorom programu. Začiatočný neterminál je <Štart>.

<Štart>::=
    `Program` <Identifikátor> `;` <Blok> `..`
<Blok>::=
    [<Definície konštánt>]
    [<Definície typov>]
    [<Definície premenných>]
    { <Definícia funkcie a podprogramu> }
    `begin` <Príkazy> `end`.

Je striktne dané poradie definícií: konštanty, typy, premenné, funkcie a procedúry (ďalej len funkcie) podľa normy. Nie sú podporované návesti (`label`) a skoky (`goto`). Programátor by mal byť schopný vytvoriť ľubovoľný program aj bez ich použitia. Výnimkou je vyskočenie z vnoreného cyklu.

V jazyku som nechal možnosť definície vložených funkcií, ktorá je v súčasnosti jedna z najviac špecifických konštrukcií pre jazyk Pascal. Každá funkcia si imaginárne vytvára (vyššiu) vrstvu identifikátorov pre konštanty, typy, premenné a funkcie. Definícia identifikátoru prekrýva identifikátor na nižšej vrstve. V jednej vrstve môže byt jeden identifikátor definovaný iba jedenkrát.

Na najnižšej globálnej vrstve sú definované identifikátory: `INTEGER`, `CHAR`, `BOOLEAN`, `REAL`, `STRING`, `PI`, `TRUE`, `FALSE`, `MAXINT`, `MININT`, `NIL`.

2.4.1. Konštanty

Je možné definovať konštanty typov `Integer`, `Boolean`, `Real`, `Char` a `String`. Je možné definovať nový identifikátor pre existujúcu konštantu (`const pi2 = pi;`). Pri definícii nových numerických konštánt je možné použiť unárne operátory plus a mínus (`const inverz_pi = – pi;`).

<Definície konštánt>::=
`Const` <Definícia konštanty> { <Definícia konštanty> }.
<Definícia konštanty>::=
<Identifikátor> = <Konštanta> `;`.
<Konštanta>::=
<Konštanta bez znamienka> |
(+ | –) (<Integer konštanta> | <Reálna konštanta> |
<Identifikátor (konštanty typu Integer alebo Real)>).
<Konštanta bez znamienka>::=
<Identifikátor (konštanty)> | <Integer konštanta> |
<Reálna konštanta> | <Textová konštanta>.

Na globálnej úrovni sú definované konštanty: `PI`, `TRUE`, `FALSE`, `MAXINT`, `MININT`, `NIL`. Definované konštanty sú skutočné konštanty, ktorých hodnoty prekladač dopĺňa počas prekladu. Nie je možné získať ich adresu v pamäti ani ich hodnotu meniť, ako je tomu napr. v prostredí Turbo Pascal [4].

2.4.2. Údajové typy

<Definície typov>::=
`Type` <Definícia typu> { <Definícia typu> }.
<Definícia typu>::=
<Identifikátor (definovaného typu)> = <Typ> `;`.
<Typ>::=
<Identifikátor (typu)> |

<Typ záznam> |
<Typ pole> |
<Typ ukazovateľ>.

Sú podporované 4 základné typy premenných. Celočíselný typ `Integer`, jeho veľkosť závisí od použitej architektúry, hraničné hodnoty sú definované v konštantách `MaxInt` a `MinInt`. Reálny typ je `Real`, jeho veľkosť je dvojnásobná oproti typu `Integer`. Ďalšími typmi sú typ `Char` pre znak s veľkosťou 1 bayt a typ `Boolean` pre pravdivostnú hodnotu. Na rozdiel od Pascalu je pretečenie hodnoty premennej považované za chybu počas vykonávania.

<Typ záznam>::=
`Record` [<Zoznam položiek> [`;`]] `End`.
<Zoznam položiek>::=
<Položky jedného typu> { `;` <Položky jedného typu>}.
< Položky jedného typu>::=
<Zoznam identifikátorov> `:` <Typ>.
<Zoznam identifikátorov>::=
<Identifikátor> { `,` <Identifikátor> }.

Typ záznam (`record`) je podporovaný. Je možné definovať typ záznam aj vo vnútri iného záznamu. Nie sú podporované tzv. variantné záznamy z Pascalu. V súčasnosti sa už nepoužívajú, pretože ich funkciu vyriešilo elegantnejšie objektové programovanie.

<Typ pole>::=
Array <Ordinálne rozsahy> Of <Typ>.
<Ordinálne rozsahy>::=
<Ordinálny rozsah> { `,` <Ordinálny rozsah> }.
<Ordinálny rozsah>::=
<Ordinálna konštanta> `..` <Ordinálna konštanta>.
<Ordinálna konštanta>::=
<Identifikátor (konštanty Integer, Char)> |
[(+ | –)] (<Identifikátor (konštanty Integer)> | <Integer konštanta>) |
`'` <Ľubovoľný znak> `'`.

Je možné definovať polia, ktoré sú indexované typom `Integer` alebo `Char`. Typ hodnôt poľa môže byť ľubovoľný. Zhustené (`packed`) polia nie sú podporované.

Reťazec (`String`) je definovaný ako `Array[ 0..255] Of Char`, pričom na nultej pozícií je uložená platná dĺžka reťazca. Reťazcový literál a premenná musia mať vždy dĺžku najviac 255 znakov, inak dôjde k chybe.

<Typ ukazovateľ>::=
    ^ <Identifikátor (typu)>.

Jazyk podporuje typ ukazovateľ. Pri definícií typu ukazovateľ cieľový typ nemusí byť definovaný, postačujúca je jeho definícia v bloku <Definície typov>. Tento stav je potrebný pre definície rekurzívnych dátových štruktúr. Napríklad lineárny spojový zoznam.

```
PNode = ^TNode;
TNode = record
      X: Integer;
      Next: PNode;
End;
```

Typ bázový ukazovateľ (`Pointer`) nie je podporovaný. Kvôli bezpečnosti nie je možné pretypovanie z jedného typu ukazovateľ na iný. Jedinou výnimkou je priradenie konštanty `Nil` do ľubovoľného ukazovateľa.

Štandardné typy jazyka Pascal množina (set) a vymenovaný typ (enumerátor) nie sú podporované. Nie je prítomný ani údajový typ súbor, použitie iného ako štandardného vstupu a výstupu nie je podporované.

2.4.3. Premenné

<Definície premenných>::=
    `Var` <Zoznam položiek> `;`.

<Zoznam položiek>::=
    <Položky jedného typu> { `;` <Položky jedného typu>}.

< Položky jedného typu>::=
    <Zoznam identifikátorov> `:` <Typ>.

Hodnota premenných pri štarte programu nie je inicializovaná.

<Premenná>::=
    <Identifikátor (premennej)> {<Pokračovanie premennej>}.

<Pokračovanie premennej>::=
    `[` <Výraz> { `,` <Výraz> } `]` |

. <Identifikátor (položky záznamu)> |
^.

2.4.4. Funkcie a procedúry

<Definícia funkcie a podprogramu>::=
Procedure <Identifikátor> [<Formálne paramatre>] ;
<Pokračovanie_funkcie> ; |
Function <Identifikátor> [<Formálne paramatre>] :
<Identifikátor (návratového typu)> ; <Pokračovanie_funkcie> ; |
Function) <Identifikátor> ; <Blok> ;.

<Pokračovanie_funkcie>::=
<Blok> |
forward.

<Formálne paramatre>::=
([<Parametre jedného typu> { ; <Parametre jedného typu> }]
).

<Parametre jedného typu>::=
<Zoznam identifikátorov> : <Identifikátor (typu)>.

Nie je podporované predávanie parametrov odkazom ako v Pascale. Ďalej nie je možné predávať súbor, pretože nie je podporovaný. Vzhľadom na stav, že jazyk bude prijímaný interpretom, je možné použiť ako návratový typ funkcií ľubovoľný typ, teda aj pole a záznam.

Je možná dopredná (forward) deklarácia funkcií. Deklarovaná funkcia musí byť definovaná v rovnakom bloku. Pri definícii sa neuvádzajú parametre ani návratový typ.

<Skutočné parametre>::=
(<Výraz> { , <Výraz> }).

Vyhodnocovanie parametrov prebieha od prvého po posledný.

2.5. Príkazy

<Príkazy>::=
[<Príkaz> { ; <Príkaz> }].
<Príkaz>::=
<Prázdny príkaz> |

(<Premenná> | <Identifikátor (aktuálnej funkcie)>) := <Výraz> |
<Identifkikátor (procedúry)> [<Skutočné parametre>] |
(Writeln | Write) [<Skutočné parametre>] |
(Readln | Read) [(<Premenná> { , <Premenná>})] |
Begin <Príkazy> End |
<Operátor premennej> (<Premenná>) |
<Príkazový symbol> |
If <Výraz> Then <Príkaz> [Else <Príkaz>] |
While <Výraz> Do <Príkaz> |
Repeat <Príkazy> Until <Výraz> |
For <Premenná> := <Výraz> (To | DownTo) <Výraz> Do
<Príkaz>.

Nie sú podporované príkazy skoku (goto), vetvenie (case) a konštrukcia with pre zjednodušenie práce s premennými záznam.

Pri priraďovaní do premennej alebo do návratovej hodnoty aktuálnej funkcie, musí byť jej typ rovnaký ako typ výrazu. Výnimkou sú dve implicitné konverzie Real:= Integer a String:= Char. Priradenie je definované pre ľubovoľný údajový typ, aj pre jednotlivé polia a záznamy.

Vstupné a výstupné operátory sú podporované pre základné typy Integer, Char, Boolean, Real a String. Je možné zadať viac parametrov oddelených čiarkami.

Neterminál <Operátor premennej> označuje operátory pre ordinálného predchodcu a následníka dec, inc, ktoré sú podporované pre typy Integer, Char a Boolean. Ďalej operátory new a dispose pre prácu s pamäťou na halde. Ich parametre musia byť premenné typu ukazovateľ. Pri použití dispose sa hodnota premennej nastaví na nil.

Neterminál <Príkazový symbol> označuje riadiace príkazy BREAK (vyskočenie z cyklu), CONTINUE (koniec časti cyklu), EXIT (ukončenie funkcie), HALT (ukončenie programu). Použitie príkazov mimo platné umiestnenie napr. break mimo cyklu je chyba.

Výraz v podmienke (if) musí byť typu Boolean.

Cykly Repeat - Until a While - Do definujem rovnako ako v štandardoch Pascalu. Nasledujúce sekvencie príkazov sú ekvivalentné:

```
Repeat
  Body;
Until condition;
```

```
Body;
Here:
If not condition then
Begin
  Body;
  goto Here;
End;
```

Podobne príkazy:

```
While Condition Do Body;
```

```
If Condition Then
Repeat
  Body
Until not Condition;
```

Pre cyklus `for` je možné ako riadiacu iteračnú premennú použiť ľubovoľnú premennú typu `Integer` a `Char`. Je možné použiť vzostupný (použitím `To`) a zostupný (`DownTo`) variant cyklu. V niektorých učebniciach Pascalu a napríklad v prostredí Borland Delphi je zmena hodnoty riadiacej premennej vo vnútri cyklu chyba pri preklade. Kvôli zložitej implementácii toto pravidlo do jazyka nezahrniem. Hodnota riadiacej premennej po ukončení cyklu nie je definovaná.

Implementáciu tohto cyklu je zložitejšia ako cyklus v jazyku C. Cyklus je definovaný nasledovne, uvedené sekvencie príkazov sú ekvivalentné.

```
For v:= initial To final Do
  Body;
```

```
temp1:= initial;
temp2:= final;
If temp1 <= temp2 Then
Begin
  v:= temp1;
  Body;
  While v <> temp2 Do
  Begin
    v:= succ( v);
```

```
    Body
  End
End
```

Rozhodol som sa použiť jednoduchšiu definíciu:

```
v:= initial;
While v <= final Do
Begin
  Body;
  inc( v);
End;
```

2.6. Výrazy

Pri vyhodnocovaní výrazov sa implicitne prevádzajú dve konverzie: hodnota `Integer` na `Real` a Znak `Char` na jednoprvkový reťazec `String`.

<Výraz>::=
<Aditívny výraz> [<Relačný operátor> <Aditívny výraz>].
<Aditívny výraz>::=
[<Un. aditívny operátor>] <Term> {<Aditívny operátor> <Term>}.
<Term>::=
<Faktor> { <Multiplikatívny operátor> <Faktor> }.
<Faktor>::=
<Konštanta bez znamienka> |
<Premenná> |
<Identifikátor (funkcie)> [<Skutočné parametre>] |
(<Výraz>) |
<Unárny operátor premennej> (<Premenná>) |
<Unárny operátor výrazu> (<Výraz>).

<Relačný operátor>::=
= | <= | >= | <> | < | >.

Relačné operátory je možné použiť pre výrazy základných typov `Integer`, `Char`, `Boolean` a `Real`. Nie sú podporované pri reťazce.

<Un. aditívny operátor>::=
+ | –.

Unárne aditívne operátory + a – je možné použiť pre výrazy numerických typov `Ineteger` a `Real`.

<Aditívny operátor>::=
+ | – | XOR | OR.

Binárne operátory + a – sú podporované pre numerické typy `Integer` a `Real`., operátor + je naviac dostupný pre zreťazenie reťazcov. Operátory XOR a OR sú podporované ako logické pre `Boolean` a binárne pre `Integer`.

<Multiplikatívny operátor>::=
* | / | DIV | MOD | AND.

Operátor * je podporovaný pre `Integer` a `Real`, Operátor / je pre typ `Real`, DIV a MOD pre `Integer`, AND logický pre `Boolen` a binárny pre `Integer`.

Operátory AND a OR sa podľa štandardov Pascalu vyhodnocujú úplne. Nakoľko sa v súčasne používaných programovacích jazykoch používa skrátené vyhodnocovanie, prevzal som riešenie z prostredia Borland Pascal. Štandardné je skrátené vyhodnocovanie, pričom použitím literálu {B+} v ľubovolnej časti kódu sa zapne a použitím {B-} sa vypne úplne vyhodnocovanie.

<Unárny operátor premennej>::=
ADDR | @ | HIGH | LOW.

Operátory ADDR a @ vracajú ukazovateľ na premennú. Operátory HIGH a LOW vracajú hraničné indexy premennej typu pole.

<Unárny operátor výrazu>::=
NOT | ODD | ABS | PRED | SUCC | INT | FRAC | TRUNC | LENGTH.

Operátor NOT neguje hodnotu `Boolean`, ODD vráti `true` pre nepárnu hodnotu `Integer`, ABS absolútnu hodnotu pre `Integer` a `Real`, PRED a SUCC ordinálneho následníka a predchodcu pre `Integer`, `Char` a `Boolean`, INT odstráni z reálneho čísla desatinnú časť, FRAC vráti desatinnú časť, TRUNC vráti celú časť desatinného čísla v celočíselnom type `Integer`, LENGTH vráti dĺžku reťazca.

Nie sú podporované operátory a funkcie: chr, ord, eof, eoln, round a pretypovania v tvare `Integer( 1.2)`.

2.7. Syntaktické diagramy

Pre reprezentáciu formálnych bezkontextových gramatík sa v učebniciach a príručkách často používajú syntaktické (koľajnicové) diagramy [6]. Vďaka rýchlejšej čitateľnosti sú pre ľudí názornejšie.

Pre každý neterminál gramatiky musí existovať jeden diagram, ktorý má vstupný a výstupný bod. Neterminály sú reprezentované hranatými a terminály zaoblenými obdĺžnikmi. Tie sú pospájané orientovanými krivkami. Diagram vyjadruje prepisovacie pravidlá gramatiky. Cesta diagramom krivkami a obdĺžnikmi predstavuje vstupné slovo. Prijímanie slova nastáva, ak existuje cesta zo vstupného do výstupného bodu.

Pre zostrojenie syntaktických diagramov som použil projekt Generation of Syntax Diagrams [7] od Prof. Fritza Jobsta a Dipl. Math. Franka Brauna z Univerzity Regensburg. Tento voľne dostupný projekt zo zadanej gramatiky vo forme niekoľkých variant Backusovej-Naurovej formy vytvára obrázky syntaktických diagramov.

Syntaktické diagramy, ktoré vygeneroval projekt pre definovanú gramatiku jazyka, sú v prílohe.

3. Používateľská príručka interpretu

V tejto kapitole sa z používateľského hľadiska rozoberú jednotlivé možnosti práce s interpretom. Používanie interpretu je jednoduché. Úlohou programu je interpretovať vstupný zdrojový kód a predchádzať možným chybám.

3.1. Spustenie interpretu

Projekt je jednoduchá konzolová aplikácia. Prvý povinný parameter je názov vstupného súboru so zdrojovým kódom. Druhým nepovinným parametrom môže byť prepínač `-i`, jeho uvedenie spôsobí výpis interných inštrukcií, do ktorých interpret preložil vstupný kód.

```
>pascallight.exe
Usage:  pascallight <input_file> [-i]
Option:
        -I   Show compiled program instructions.
```

Prácu interpretu demonštruje použitie programu, ktorý vypíše text „Ahoj!“.

Obsah súboru ahoj.pas:

```
Program Ahoj;

begin
  writeln( 'Ahoj!');
end.
```

Konzolový príkaz:

```
>pascallight ahoj.pas
```

Príkaz spôsobí výpis niekoľkých informácií o priebehu prekladu programu a text „Ahoj!“:

```
Compiling...
Compiling OK
Running:
Ahoj!
```

V prípade zadania prepínača `-i` budú vypísané interné inštrukcie interpreta. V tomto prípade sú vypísané 3 inštrukcie hlavného bloku programu: Načítanie reťazcového literálu „Ahoj!“, výpis reťazca a výpis nového riadku.

Výpis inštrukcií:

```
Compiled instructions:
Main Block: (layer: 0)
0: Load Literal  Ahoj!
1: Write String
2: Write New Line
```

Niekoľko testovacích súborov so zdrojovými kódmi sa nachádza v prílohe.

3.2. Chyby počas prekladu

Interpret prijíma práve jazyk Pascal light, ktorého lexikálne, syntaktické a sémantické pravidlá boli definované v predchádzajúcej kapitole. V prípade nájdenia odlišnosti vypíše projekt chybu. Chybný program nebude interpretovaný. Následne projekt pokračuje v rozpoznávaní zdrojového kódu a snaží sa nájsť ďalšie chyby.

Ukážka chybného zdrojového kódu:

```
Program ChybaPreklad;
var
  x: integer;
begin
  x:= 'ahoj';
  ahoj:= 1;
end.
```

Interpret vypíše zoznam nájdených chýb s ich popisom a pozíciou:

```
Some errors found:
Error: Incompatible type of parameter or
assignes value, line 5
>  x:= 'ahoj';
    ^
Error: Unknown identifier, line 6
>  ahoj:= 1;
   ^
```

V nasledujúcich podkapitolách uvediem prehľad jednotlivých chýb, ktoré interpret rozpoznáva počas prekladu. Chyby sú rozdelené podľa typu na lexikálne, syntaktické a sémantické.

3.2.1. Lexikálne chyby

Počas lexikálnej analýzy rozpoznáva program nasledujúce chyby:

- neznámy znak - výskyt znaku, ktorý nie je obsiahnutý v žiadnom literáli mimo reťazca
- reťazec nebol ukončený pred koncom riadku alebo súboru
- reťazec je príliš dlhý
- komentár nebol ukončený
- neočakávaný znak konca komentára
- zlý formát alebo hodnota mimo rozsahu pre celočíselnú alebo reálnu konštantu

3.2.2. Syntaktické chyby

Na rozpoznávanie gramatiky jazyka som použil syntaktický analyzátor GNU Bison [9]. Výpis syntaktických chýb je preto odlišný. Syntaktická chyba nastáva pri načítaní lexikálneho žetónu (token), ktorý nebol neočakávaný. V tej situácii vypíše interpret chybu o neočakávanom žetóne a vypíše žetóny, ktoré by podľa definovanej gramatiky jazyka očakával.

Nasledujúca sekvencia je syntakticky nesprávna. Namiesto príkazu `end` bol očakávaný `until` na ukončenie cyklu alebo iný príkaz:

```
Repeat
  prikaz;
end;
```

Interpret vypísal chybu. Očakával `until` alebo bodkočiarku (prázdny príkaz):

```
Some errors found:
syntax error, unexpected TOKEN_END, expecting
TOKEN_UNTIL or TOKEN_SEMICOLON, line 7
>  end ...
   ^
```

3.2.3. Sémantické chyby

Počas sémantickej analýzy rozpoznáva program nasledujúce chyby:

- použitie nedefinovaného identifikátora
- duplikovaná definícia identifikátora v danej vrstve
- identifikátor nie je údajový typ, konštanta, procedúra, funkcia
- vstup / výstup iného typu ako `Integer`, `Char`, `Real` alebo `Boolean`

- nekompatibilný operátor / priradenie pre uvedené typy
- typ podmienky nie je typu `Boolean`
- nesprávne použitie operátorov `^`, `new`, `dispose`, typ nie je ukazovateľ
- riadiaca premenná cyklu `for` nie je ordinálna
- typ výrazu je odlišný ako typ riadiacej premennej cyklu
- premenná použitá operátormi `inc` / `dec` nie je ordinálna
- použitie riadiacich príkazov `break`, `continue`, `exit` mimo cyklu / funkcie
- nesprávne použitie identifikátora pre návratovú hodnotu funkcie
- nesprávny počet parametrov funkcie
- použitie identifikátoru typu vo výraze
- použitie operátoru bodka, premenná nie je záznam
- identifikátor nie je položka premennej záznam
- nesprávna definícia indexu poľa - zlý typ indexu poľa, dolná hranica poľa je väčšia ako horná hranica
- použitie operátoru `[]` - premenná nie je pole, odlišný typ
- deklarovaná funkcia nebola definovaná
- nepovolené definovanie parametrov pri definícii dopredne deklarovanej funkcie

3.3. Chyby počas interpretácie - základné chyby

Kľúčovým prínosom práce je odhaľovanie chýb počas interpretácie programu. Pred vykonaním jednotlivých inštrukcií musí projekt preveriť, či daná inštrukcia nespôsobí pád programu. Až následne dôjde buď k vykonaniu inštrukcie alebo hláseniu o chybe. Nie je prípustné, aby sa interpret pokúsil vykonať inštrukciu, ktorá spôsobí pád interpreta.

V nasledujúcom príklade demonštrujem spôsob výpisu chyby pri jednoduchom programe, ktorý obsahuje delenie nulou:

```
x:= 1;
dec( x);
writeln( 2 div x);
```

Interpret vypisuje hlásenie o chybe v tomto tvare:

```
Running:
Runtime Error: Integer division by Zero, line 9,
column 14
```

V nasledujúcich podkapitolách uvediem prehľad základných chýb, ktorých vznik interpret preveruje. Vyložím aj spôsob ich odhaľovania.

3.3.1. Delenie nulou

Predchádzanie delení nulou je triviálne. Interpret pred každou operáciou z kategórie: celočíselné delenie (`div`), reálne delenie (`/`) a zvyšok po delení (`mod`) preverí, či hodnota deliteľa nie je nula.

3.3.2. Neinicializovaná hodnota premennej

Podobne ako v jazyku Pascal, nie sú hodnoty premenných pri štarte programu alebo alokácii inicializované. Čítanie hodnoty neinicializovanej premennej je považované za chybu. Každá premenná má príznak inicializovanej hodnoty premennej.

Príklad chyby s neinicializovanou hodnotou. Priradenie `b:= a;` spôsobí, že hodnota premennej `b` je neinicializovaná. Pri vyhodnocovaní operácie `b + 1` nastáva chyba, pretože hodnota premennej b nie je inicializovaná:

```
b:= 2; {inicializácia premennej b}
b:= a; {priradenie neinicializovanej hodnoty}
writeln( b + 1); {chyba}


> Runtime Error: Reading of Uninitialized
Variable, line 8, column 14
```

3.3.3. Pretečenie definičného oboru údajových typov

Na rozdiel od štandardného Pascalu je pretečenie a podtečenie hodnôt premenných a výrazov považované za chybu.

Nasledujúce príkazy skončia chybou:

```
writeln( MaxInt); {Najvyššia definovaná hodnota}
x:= 1;
while true do
begin
  x:= 10 * x;
  write( x, ', ');
end;

> 2147483647
```

```
> 10, 100, 1000, 10000, 100000, 1000000,
10000000, 100000000, 1000000000,
> Runtime Error: Integer value is Too Big, Out
of Range, line 11, column 12
```

Interpret ošetruje chyby, ktoré vznikajú pri pretečení a podtečení hodnôt údajových typov `integer`, `Real`, `Char` a `Boolean`.

Predchádzanie týmto chybám je implementované známym spôsobom, ktorý prezentuje napríklad pán Virius [10]. Napríklad pred sčítaním dvoch hodnôt typu `Integer` v tvare `a + b` otestujem podmienku pre indikáciu pretečenia: `a > MaxInt – b`. V prípade pretečenia sa vypíše chyba.

3.3.4. Kontrola indexov polí a reťazcov

Pri každom prístupe k premennej typu pole resp. reťazec pomocou operátora hranaté zátvorky (`[]`) dochádza ku kontrole hraníc indexu. Ďalej dochádza ku kontrole pretečenia dĺžky reťazca pri zreťazení pomocou operátora +.

Spôsob výpisu týchto chýb:

```
> Array Index is Out of Range, line 39, column
24
> String is Too Long, line 13, column 11
```

3.4. Chyby počas interpretácie - práca s ukazovateľmi

Práca s ukazovateľmi a alokáciou premenných na hlade spôsobuje programátorom najväčšie problémy. Odhaľovanie týchto chýb je zrejme najväčší prínos tejto práce. Mojím cieľom nebolo implementovať automatickú správu haldy. Jej správu som nechal na programátorovi implementáciou operátorov `new` a `dispose`. Interpret testuje pri každej operácii s ukazovateľom jej korektnosť.

3.4.1. Implementácia overovania chýb pri práci s ukazovateľmi

Kvôli kontrole práce s ukazovateľmi obsahuje každá vytvorená premenná nasledujúce informácie. Pre položky štruktúrovaných premenných (záznam a pole) sú tieto informácie spoločné pre všetky položky:

- príznak, či je premenná umiestnená na zásobníku alebo na halde
- zoznam premenných typu ukazovateľ, ktoré referujú na danú premennú

Každá premenná typu ukazovateľ obsahuje tieto informácie:

- príznak inicializácie ukazovateľa - premennej bola priradená hodnota
- hodnota - adresa premennej, na ktorú smeruje ukazovateľ
- príznak platnosti adresy - v prípade uvoľnenia premennej sa príznak ruší

Na nasledujúcom príklade postupne demonštrujem nastavovanie a prácu s týmito príznakmi:

```
Var
  a: Integer;
  p, r: ^Integer;
```

Hodnota ukazovateľa p pri štarte programu nie je inicializovaná. Čítanie jej hodnoty spôsobí chybu:

```
if p = Nil then writeln( ‘Nil’);

> Runtime Error: Reading of Uninitialized
Variable
```

Pri priradení adresy premennej a do ukazovateľa p sa pridá ukazovateľ p do zoznamu ukazovateľov, ktoré referujú na premennú a:

```
p:= Addr( a);
```

Po priradení inej hodnoty do ukazovateľa p sa ukazovateľ p odstráni zo zoznamu ukazovateľov na premennú a:

```
new( p);
```

Premenná je alokovaná na halde. Pri priradení ukazovateľa p do ukazovateľa q, bude alokovaná premenná obsahovať dvojprvkový zoznam referujúcich ukazovateľov:

```
q:= p;
```

Pri uvoľnení premennej z haldy sa prejde zoznam referujúcich ukazovateľov. Každému ukazovateli sa zruší príznak platnosti adresy. Kvôli tomu dereferencovanie ukazovateľa p spôsobí chybu.

```
dispose( q);
p^:= 100;

> Runtime Error: Bad Access to pointer -
Variable has been disposed
```

V nasledujúcich podkapitolách uvediem možné chyby pri práci s ukazovateľmi, ktoré interpret odhaľuje. Tieto chyby demonštrujem na príkladoch, ktoré sú uvedené aj v priloženom CD médiu.

3.4.2. Neinicializovaný ukazovateľ

Podobne ako pri premenných základných typov, hodnota ukazovateľa pri štarte resp. alokácii premennej nie je inicializovaná. Jeho čítanie, porovnávanie jeho hodnoty alebo dereferencia je považované za chybu.

```
Begin
  writeln( k^); {ukazovatel nebol
inicializovany}
end.
> Runtime Error: Reading of Uninitialized
Variable
```

3.4.3. Dereferencia ukazovateľa s hodnotou Nil

Nie je možné dereferencovať ukazovateľ natavený na `Nil`.

```
Begin
  l:= nil;
  writeln( l^); {CHYBA, l^ - pristup k adrese
Nil}
end.
```

3.4.4. Dereferencia ukazovateľa s neplatným príznakom platnosti

Najčastejšia a najhoršie detekujúca chyba nastáva, ak chce programátor používať ukazovateľ, ktorý odkazuje na premennú, ktorá už bola odstránená z pamäte. Sú dve možnosti vzniku. Buď ukazovateľ referoval na lokálnu premennú na zásobníku alebo bola premenná operátorom `dispose` uvoľnená z haldy.

Pri uvoľnení premennej sa prejde zoznam referujúcich ukazovateľov a každému ukazovateli sa zruší príznak platnosti adresy. Pri dereferencovaní neplatného ukazovateľa vypíše interpret chybu:

A) Neplatná adresa lokálnej premennej:

```
Var
  k: ^Integer;

procedure parameter_reference( a: integer);
begin
  k:= @a; {priradenie adresy lokalnej premennej}
  writeln( 'Parameter: ', a);
end;

begin
  parameter_reference( 123);
  writeln( k^); {CHYBA, lok.premenna bola
uvolnena}
end.

> Runtime Error: Bad Access to pointer -
Variable has been disposed,
```

B) Premenná bola odstránená z haldy:

```
Var
  k, l: ^Integer;

begin
  new( k);
  l:= k;
  l^:= 246;
  l^:= k^ div 2;

  writeln( 'k^ = ', k^);
  dispose( l);
  writeln( 'k^ = ', k^);
end.
> k^ = 123
> k^ = Runtime Error: Bad Access to pointer -
Variable has been disposed,
```

Používané prekladače, ktoré nepoužívajú automatickú správu haldy túto chybu neošetrujú. Ak ukazovateľ referuje na pamäť, ktorá už je uvoľnená, dostane programátor chybové hlásenie operačného systému (správy virtuálnej pamäte) o nepovolenom prístupe k pamäti.

Ešte zložitejšia situácia nastáva, ak program daný blok pamäte opätovne alokuje. Programátor sa tak o chyba nedozvie. O tejto situácii zaujímavo píše pán Virius [10]:

„Volná paměť počítače, kterou může program použít pro alokaci dynamických proměnných, může být organizována jako spojový seznam bloků jisté standardní velikosti. Pokud se při uvolnění části paměti programem bude vracet celý blok operačnímu systému, zařadí se do tohoto seznamu, a to znamená, že se část jeho obsahu přepíše ukazatelem na další volný blok. Poznamenejme, že ve starších verzích Windows jsme při alokaci mohli dostat zpět právě uvolněný blok paměti, neboť uvolněné bloky se vracely na počátek seznamu volné paměti - seznam volné paměti se choval v podstatě jako zásobník. Od verze Windows 98 se však uvolněná paměť vrací na konec seznamu volné paměti – seznam volné paměti se chová jako fronta. To vede k rychlejšímu odhalení chyb tohoto duhu – a vedlo to také ke zhroucení řady programů, které ve starších verzích Windows zdánlivě fungovaly.“

3.4.5. Nesprávne použitie operátora dispose

Operátor `dispose` sa používa na uvoľnenie pamäte na halde. Uvoľňovaná pamäť sa musí nachádzať na halde a musí to byť vrchol alokovaného bloku. Po uvoľnení sa premenná ukazovateľa nastaví na `Nil`. Kompilátory programovacích jazykov, v ktorých sa o uvoľnenie pamäte musí starať programátor, nemôžu kontrolovať správnosť, pretože tú riadi správa virtuálnej pamäte.

Pri použití operátora `dispose` môžu vzniknúť 4 druhy chýb, druhé dve chyby sú špecifické výhradne pre operátor `dispose`:

A) Použitie `dispose` na ukazovateľ s hodnotou `Nil`.

```
pr:= Nil;
dispose( pr); {CHYBA - dispose Nil}
> Runtime Error: Access to pointer with Nil
Value
```

B) Použitie na pamäť, ktorá bola medzitým uvoľnená.

```
new( pr);
ps:= pr;
dispose( ps);
{dispose( pr); {CHYBA - premenna uz bola
uvolnena}
> Runtime Error: Bad Access to pointer -
Variable has been disposed
```

C) Použitie na pamäť, ktorá sa nachádza na zásobníku:

```
pr:= Addr( r);
dispose( pr); {CHYBA - premenna je na zasobniku}
> Runtime Error: Illegal Dispose - Memory is
Stack variable
```

D) Operátor `dispose` sa musí použiť na uvoľnenie celého bloku pamäte. V prípade, že bola pamäť alokovaná pre štruktúrovanú premennú typu pole alebo záznam, nemôže byť dealokovaná iba jedna položka poľa, záznamu:

```
new( pr);
px:= Addr( pr^.a);
dispose( px); {CHYBA - Uvolnenie polozky
zaznamu}

new( p);
px:= Addr( p^[ 1]);
dispose( px); {CHYBA - Uvolnenie polozky pola}

> Runtime Error: Illegal Dispose - Memory is
only Part of Record or Array
```

3.4.6. Nedostupná pamäť na halde

Chyba, ktorá sa pri programovaní malých úloh a projektov vôbec nemusí prejaviť, je neuvoľnenie dynamicky alokovanej pamäte. Táto chyba sa neprejaví pádom programu. Jej následky sú viditeľné až pri dlhšom behu programu a výraznej spotrebe pamäte systému. Táto chyba je známa pod anglickým názvom Memory leak.

Príklad vzniku nedostupnej premennej:

```
Type PInteger = ^Integer; PPInteger = ^PInteger;
Var k: PInteger; pp: PPInteger;
```

```
begin
  new( k); {k -> halda}
  k^:= 123;
  writeln( 'k^ = ', k^);

  new( pp); {pp -> halda}
  pp^:= k;  {pp -> halda -> halda}
  k:= nil;
  writeln( 'pp^^ = ', pp^^);

  {pp -> X; ? -> halda}
  pp^:= nil; {CHYBA - nedostupna pamat}
end.

> k^ = 123
> pp^^ = 123
> Runtime Error: Inaccessible Heap memory,
```

Odhaľovanie nedostupných blokov pamäte na halde funguje vďaka počítaniu referencií. Každá premenná (blok pamäte) obsahuje počet a zoznam premenných, ktoré na ňu referujú. Táto metóda je pomalá, pretože pri každom príkaze priradenia hodnoty ukazovateľa sa musí prevádzať niekoľko inštrukcií. Výhodou tejto metódy je, že nedostupnosť detekuje hneď pri jej vzniku. Úlohou tejto práce je upozorňovať programátora na chyby hneď pri ich vzniku, preto som použil túto metódu.

Ako píše Bruce Eckel [11], metóda počítania referencií nedokáže odhaliť nedostupnosť pamäte, ak z referujúcich premenných vznikne usporiadaný cyklus. To môže nastať aj pri jednoduchom lineárnom cyklickom spojovom zozname. Zostrojený interpret nie je schopný odhaliť tento stav. Možnou nápravou by bolo použiť grafový algoritmus pre hľadanie cyklov.

Metóda počítania referencií sa kvôli spomenutým nedostatkom nepoužíva pri implementácii automatickej správy haldy pomocou zberača nedostupných objektov. Platformy Java a .NET uprednostňujú občasné pozastavenie programu a zložitejšie kombinované postupy pre hľadanie nedostupných blokov pamäte.

4. Popis implementácie

4.1. Použité technológie

Pri implementácii interpretu boli pre mňa cenné poznatky z teórie prekladačov. Ako programovací jazyk som zvolil jazyk C++ kvôli vysokej dostupnosti generátorov lexikálnej a syntaktickej analýzy.

Zdrojový kód neobsahuje žiadne neštandardné a neprenositeľné knižnice, interpret je možné skompilovať na každej platforme s existujúcim prekladačom jazyka C++. Ako vývojové prostredie som použil Microsoft Visual Studio 2005. Pre zostrojenie lexikálneho analyzátora som použil GNU Flex 2.5.4a, pre vytvorenie syntaktického analyzátora GNU Bison 2.1. Dokumentácia vznikla použitím Doxygen 1.5.5. Zdrojový kód interpretu a vygenerovaná dokumentácia sa nachádzajú v priloženom CD - médiu.

4.2. Lexikálna a syntaktická analýza

Pri tvorbe prekladačov formálnych jazykov sa používajú nástroje pre tvorbu lexikálnych a syntaktických analyzátorov. Tieto nástroje zo vstupných pravidiel vygenerujú zdrojový kód v programovacom jazyku.

Pravidlá lexikálnej analýzy sa nachádzajú v súbore `flex.lex`. Nástroj Flex podľa nich vygeneruje zdrojový kód. Úlohou lexikálnej analýzy je zo vstupného kódu vytvoriť postupnosť literálov (základné slovo jazyka). Spolu s niektorými literálmi vracia aj ich prídavnú informáciu. Napríklad pri celočíselnej konštante jej hodnotu. Lexikálne pravidlá sú zapísané pomocou regulárnych výrazov. Kompilátor podľa nich zostrojí konečný automat pre rozpoznávanie literálov.

Identifikátory a reťazcové konštanty sa ukladajú do literárnych tabuliek. Prekladač v ďalších fázach nemusí pracovať s textom, ale iba s ukazovateľmi do tabuliek.

V súbore `bison.y` je zápis gramatiky jazyka v Backusovej - Naurovej forme. Podľa toto súboru vytvorí nástroj Bison syntaktický analyzátor. Jeho úlohou je rozpoznať, či zápis program zodpovedá požadovanej gramatike a vytvoriť syntaktický strom podľa logickej štruktúry programu.

4.3. Sémantická analýza a generovanie inštrukcií

Úlohou sémantickej analýzy je spracovať syntaktický strom a preveriť správnosť operácií. Do sémantických tabuliek sa ukladajú definície konštánt, typov, premenných, funkcií a procedúr. Pre príkazovú časť kódu prevádza typovú kontrolu, konverzie a generuje inštrukcie pre vstupné príkazy a výrazy.

Tieto definície sa ukladajú do asociatívneho kontajnera indexovaného identifikátormi. Jazyk podporuje vnorené funkcie a procedúry (ďalej len funkcie). Preto sa pre každej ďalšej definícii vnorenej funkcie vytvára vrstva identifikátorov. V danej vrstve môže byť jeden identifikátor použitý iba jedenkrát. Použitie identifikátora zatieňuje identifikátor v nižších vrstvách. V C++ som tieto vrstvy reprezentoval kontajnerom `std::vector< std::map< Identifikátor, Referencia na definíciu >>`. Pri použití definovaného symbolu (napríklad premennej) v príkazovej časti sa identifikátor hľadá od najvyššej vrstvy. Každá premenná a funkcia obsahuje v sémantických tabuľkách aj informáciu o vrstve, v ktorej je umiestnená.

4.4. Inštrukcie

Na rozdiel od bežných prekladačov netransformujem kód do lineárnej postupnosti inštrukcií, ale do rekurzívnej stromovej štruktúry. Každej funkcii, resp. hlavnému bloku programu zodpovedá v strome inštrukcií jeden vrchol. Tento vrchol obsahuje lineárnu postupnosť inštrukcií tela danej funkcie a odkazy na vrcholy, ktoré zodpovedajú vnoreným funkciám.

Inštrukcie, do ktorých sa transformuje vstupný zdrojový kód sú reprezentované hierarchickou štruktúrou tried so spoločným predkom. Na druhej úrovni dedičnosti sa inštrukcie rozdeľujú podľa ich vzťahu k zásobníku. Inštrukcie, ktoré majú rovnaký prístup k zásobníku majú aj rovnaké rozhranie. Sú definovaní predkovia pre inštrukcie unárnych, binárnych operátorov, inštrukcie, ktoré pridávajú alebo odoberajú premenné zo zásobníku, špeciálne príkazy a podmienené skoky.

Pre vytvorenú inštrukciu sa vytvára inštancia triedy reprezentujúcej inštrukciu. Vykonanie správnej inštrukcie zabezpečuje volanie virtuálnych metód. Detekcia jednotlivých chýb je obsiahnutá v kóde metód tried pre inštrukcie.

4.5. Vykonávanie inštrukcií

Trieda `LangInterpret` prijíma inštrukcie v stromovej štruktúre a vykonáva ich. Interpretácia inštrukcií začína v hlavnom bloku. Ak inštrukcia nie je skok, tak sa pokračuje vo vykonávaní v nasledujúcej inštrukcii. Pri volaní a ukončovaní funkcie sa ukazovateľ na vykonávanú inštrukciu vertikálne pohybuje v strome inštrukcií.

Súčasť interpreta je zásobník premenných, v ktorom sú umiestnené premenné, parametre funkcie a dočasné výsledky pri vyhodnocovaní výrazov. Zásobník je typovaný pre údajové typy premenných tak, že jednej premennej zodpovedá jedna položka zásobníku. Aj polia a záznamy sú reprezentované ako jedna (vnútri štruktúrovaná) položka. Zásobníku je dynamické pole (`std::vector`) typu ukazovateľ na abstraktného predka premenných. Od tohto predka existujú potomkovia pre premenné základných typov, polí a záznamov.

V implementácii nie je takmer žiadna odlišnosť medzi premennými, ktoré sú umiestnené na zásobníku a na halde. Obidva druhy premenných sú implementačne alokované halde.

V zásobníku volaní funkcií (Call stack) je pre každú zavolanú vnorenú funkciu uložená pozícia na zásobníku, ktorá zodpovedá umiestneniu lokálnych premenných danej funkcie.

5. Záver

Mojím cieľom bolo vytvoriť jednoduché vývojové prostredie vo forme interpreta, ktoré bude detekovať niektoré programátorské chyby. Najväčšou motiváciou bolo zvládnutie chýb pri práci s ukazovateľmi a dynamicky alokovanými premennými.

Nebolo cieľom implementovať celý jazyk, ale zamerať sa pri implementácii najmä na odhaľovanie chýb. Neočakávam použitie projektu v komerčnej sfére, interpret je vhodný napríklad pri výučbe programovania alebo otestovaní práce jednoduchých programov. Preto som si vybral programovací jazyk Pascal. V kapitole 2 som našpecifikoval redukciu jazyka Pascal, ktorý bude interpret rozpoznávať a vykonávať.

Podarilo sa mi vytvoriť jednoduché vývojové prostredie spúšťané z príkazového riadku. Projekt interpretuje špecifikovaný jazyk, podporuje vnorené funkcie a prácu s ukazovateľmi. Hlavný prínos interpretu je v detekcii programátorských chýb, ktoré sa prejavujú nepovolenými operáciami. Projekt kontroluje chyby pri práci s neinicializovanými premennými, aritmetickými chybami a nesprávnom prístupe k premenným typu pole.

Oproti iným prostrediam je výhodné použiť interpret pri práci s dynamickými premennými. Nie je implementovaná automatická správa haldy, prácu s alokáciou ponecháva projekt na programátorovi. Úlohou interpretu je preverovať, či programátor správu haldy zvládol správne naprogramovať. Kontroluje napríklad správnosť dereferencií ukazovateľov a dostupnosť alokovaných premenných na halde.

Myslím si, že môj cieľ sa mi podarilo dosiahnuť. Pri špecifikácii jazyka, implementácii programu a písaní tejto práce som sa veľa naučil. Po ukončení implementácie som mal pocit, že som možno mohol podporovaný programovací jazyk ešte viac redukovať a zamerať na detekciu ďalších chýb. Implementácia vnorených funkcií bola zložitá a zaujímavá, ale používateľ interpreta ju nemusí výrazne oceniť. Splnil som však svoj hlavný cieľ, ktorým bola práca s ukazovateľmi. V téme detekcii programátorských chýb sa vždy dá pokračovať. Nevýhodou interpretu je neobsiahnutie používateľského rozhrania. Programátor sa dozvie o svojich chybách, ale nájdenie presnej príčiny v zdrojovom kóde je na programátorovi.

Literatúra

[1] Wirth N. E.: Životopis
http://www.cs.inf.ethz.ch/~wirth/

[2] Kolektív (1990): Štandard jazyka Pascal ISO/IEC 7185:1990
http://www.pascal-central.com

[3] Bret F.: IPcute
www.ipcute.free.fr

[4] Mikula P., Juhová K., Soukenka J. (1993): Turbo Pascal 7.0 Kompletní průvodce, Grada

[5] Garshol L. M.: BNF and EBNF: What are they and how do they work?
http://www.garshol.priv.no/download/text/bnf.html

[6] Prispievatelia Wikipédie: Syntaktické diagramy
http://en.wikipedia.org/wiki/Syntax_diagram

[7] Jobst F., Braun F.: Generation of Syntax Diagrams
http://www-cgi.uni-regensburg.de/~brf09510/syntax.html

[8] Dokumentácia GNU Flex
http://www.gnu.org/software/flex/manual/html_mono/flex.html

[9] Dokumentácia GNU Bizon
http://www.gnu.org/software/bison/manual/html_mono/bison.html

[10] Virius M. (2005): Pasti a propasti jazyka C++, CP Books

[11] Eckel B. (2001): Myslíme v jazyku Java : knihovna programátora, Grada

[12] Yaghob J.: Prezentácia k prednáške Principy překladačů
http://ulita.ms.mff.cuni.cz/pub/predn/pp

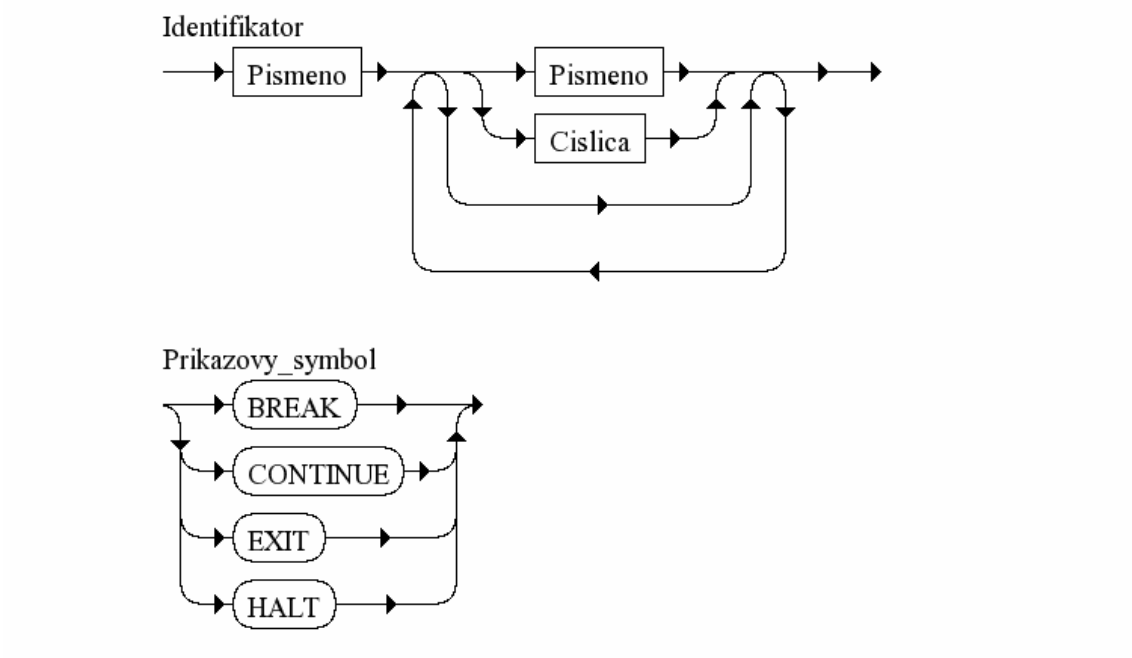
[13] Aho A. V., Sethi R., Ullman J.D. (1986): Compilers Principles, Techniques, and Tools, Addison Wesley Longman

Prílohy

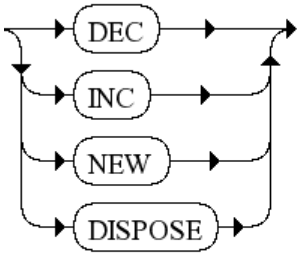
V priloženom CD - médiu sa nachádza text práce a interpret s testovacími zdrojovými súbormi.

Syntaktické diagramy

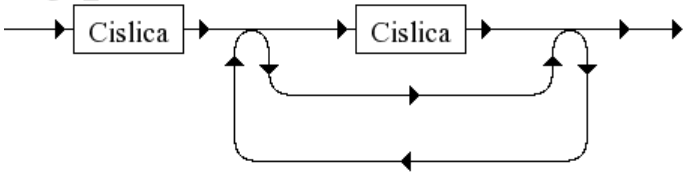
Pre gramatiku jazyka Pascal light vygeneroval projekt Generation of Syntax Diagrams [7] nasledujúce syntaktické diagramy.



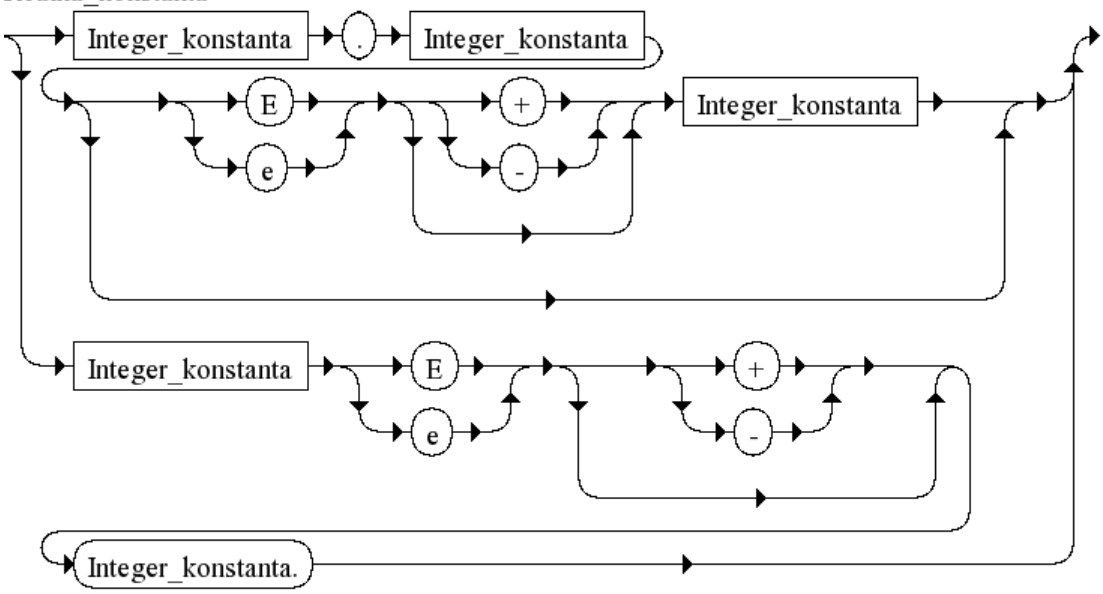
Operator_premennej



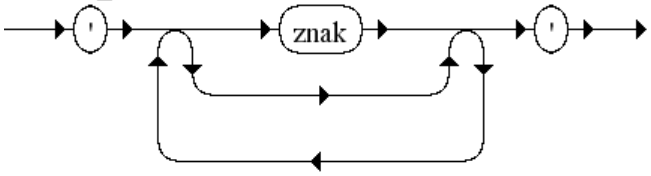
Integer_konstanta



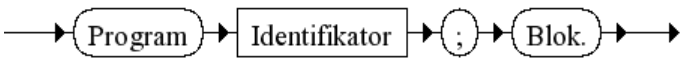
Realna_konstanta



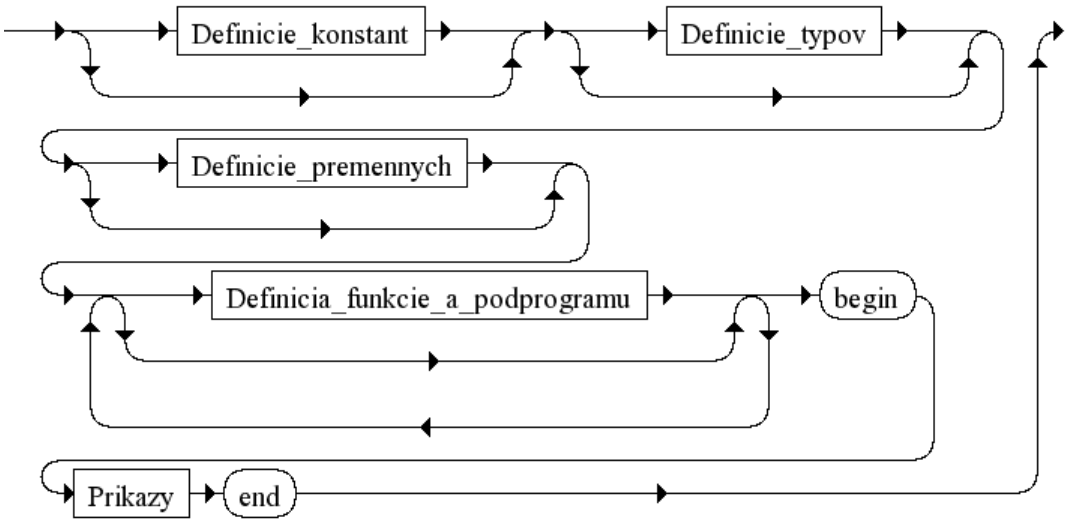
Textova_konstanta



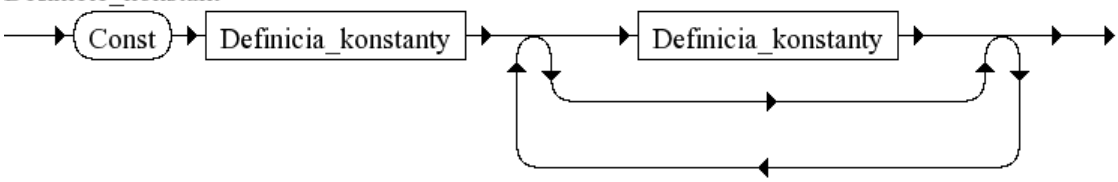
Start



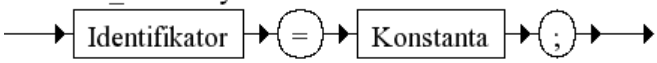
Blok



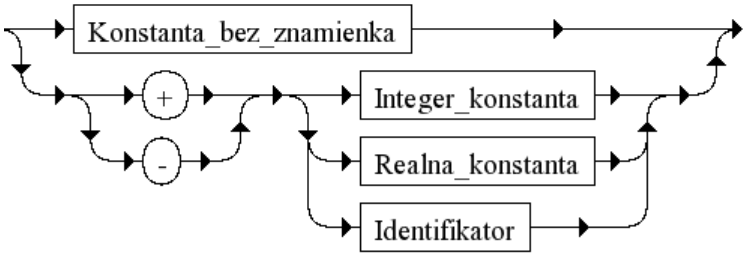
Definicie_konstant



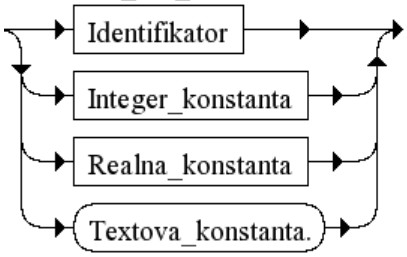
Definicia_konstanty



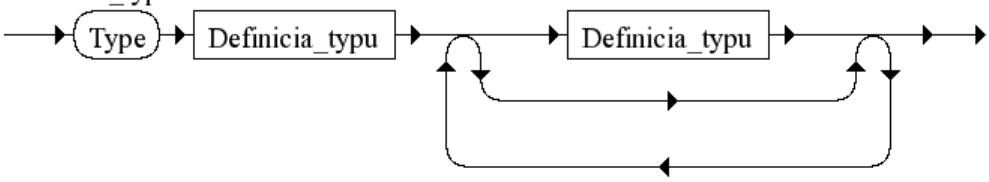
Konstanta



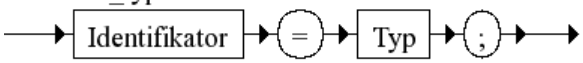
Konstanta_bez_znamienka



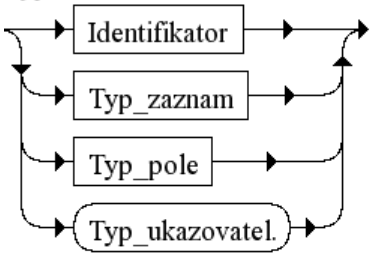
Definicie_typov



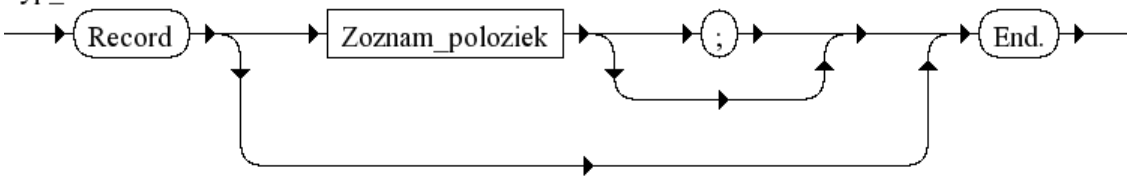
Definicia_typu



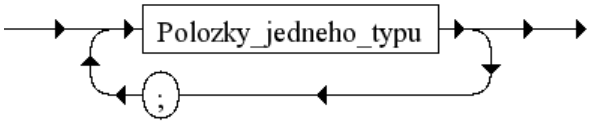
Typ



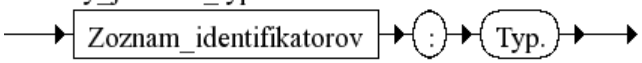
Typ_zaznam



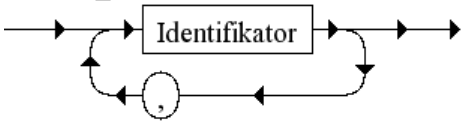
Zoznam_poloziek



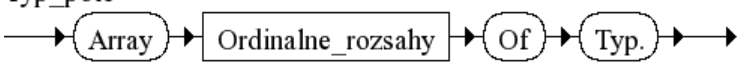
Polozky_jedneho_typu



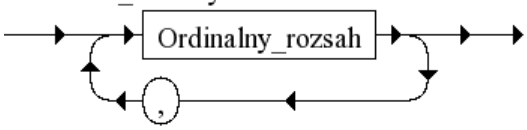
Zoznam_identifikatorov



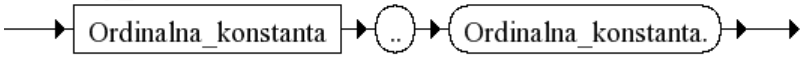
Typ_pole



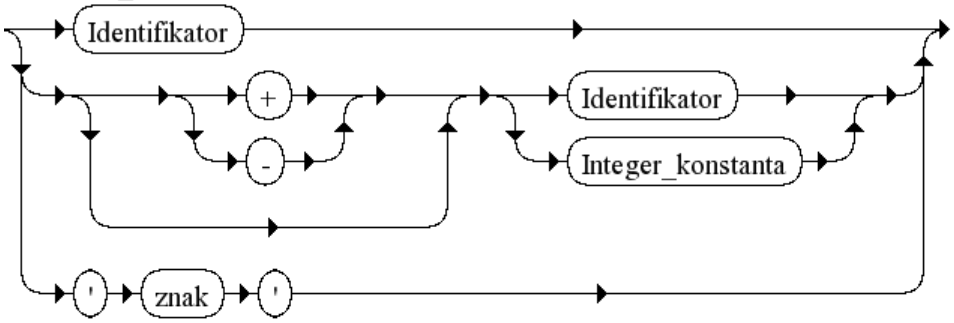
Ordinalne_rozsahy



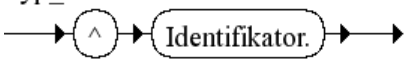
Ordinalny_rozsah



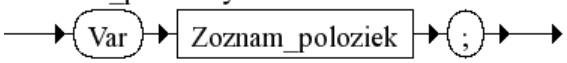
Ordinalna_konstanta



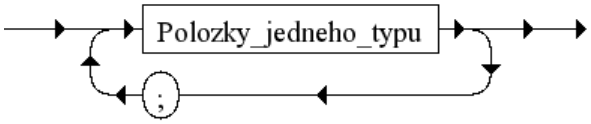
Typ_ukazovatel



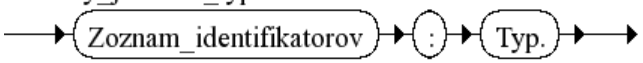
Definicie_premennych



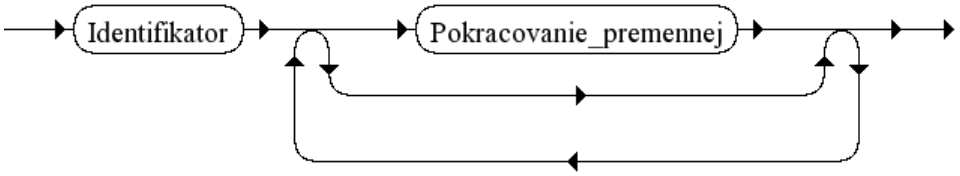
Zoznam_poloziek



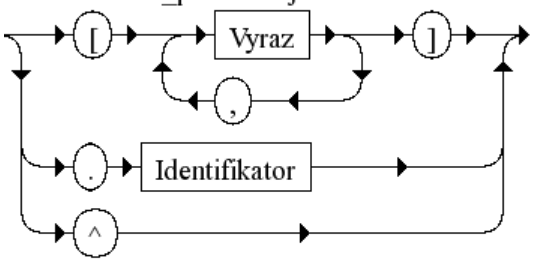
Polozky_jedneho_typu



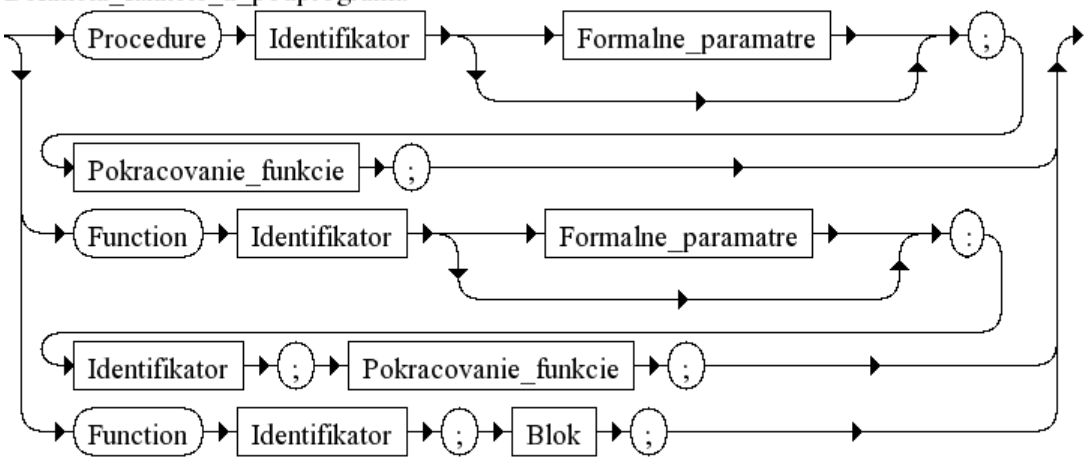
Premenna



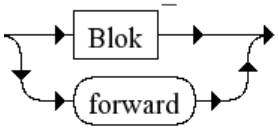
Pokracovanie_premennej



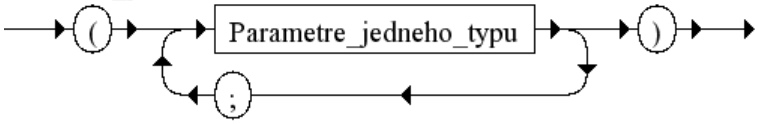
Definicia_funkcie_a_podprogramu



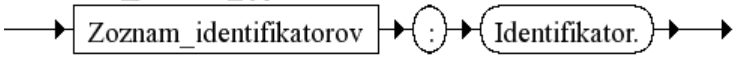
Pokracovanie_funkcie



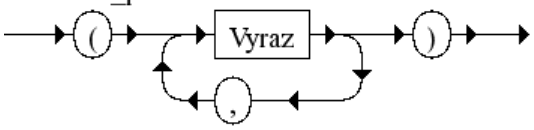
Formalne_paramatre



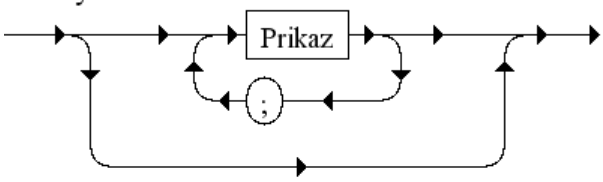
Parametre_jedneho_typu



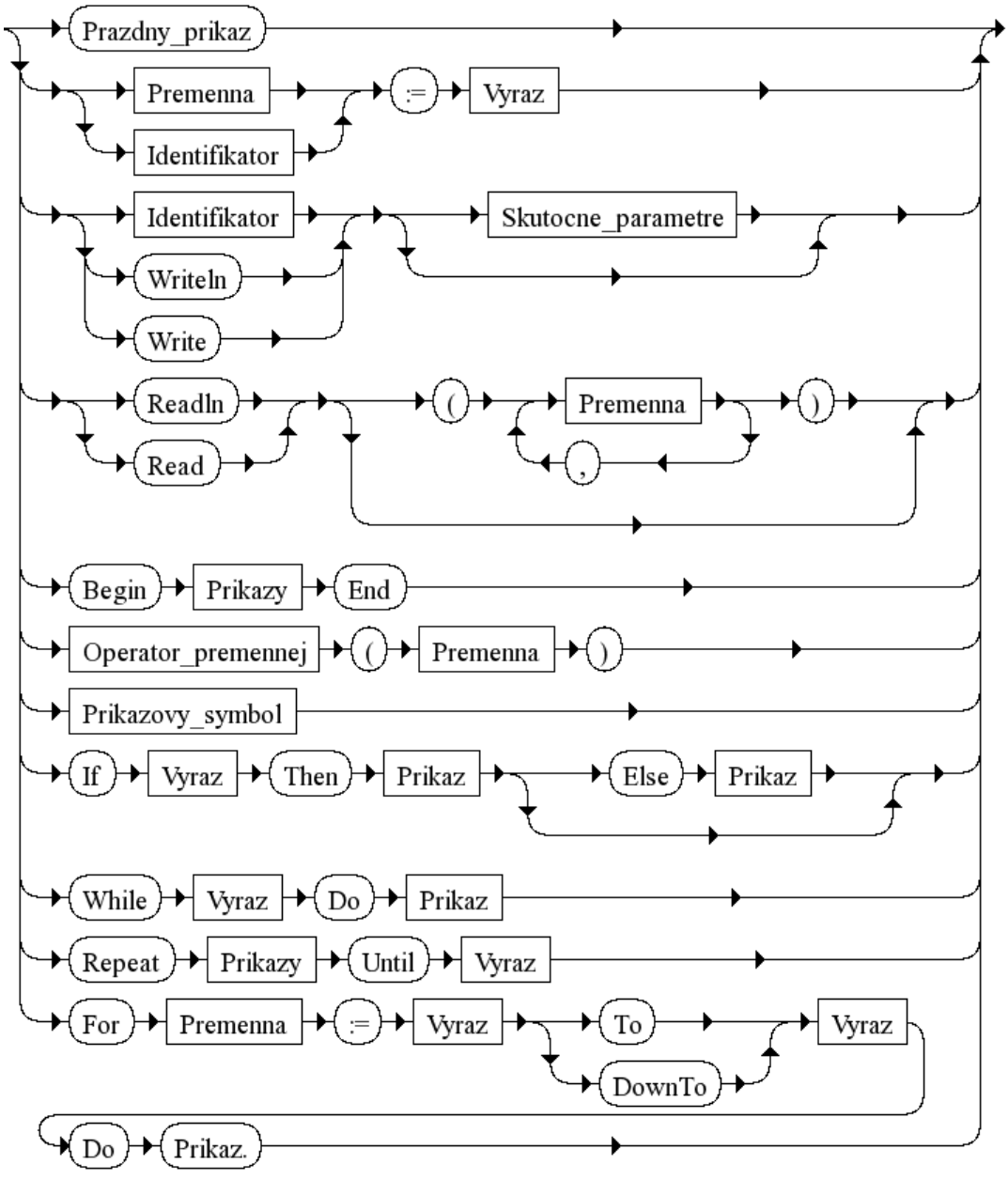
Skutocne_parametre



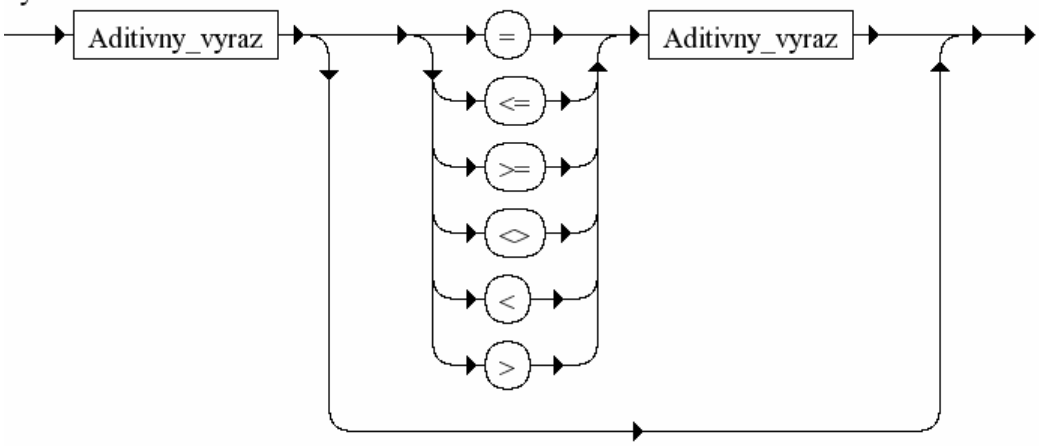
Prikazy



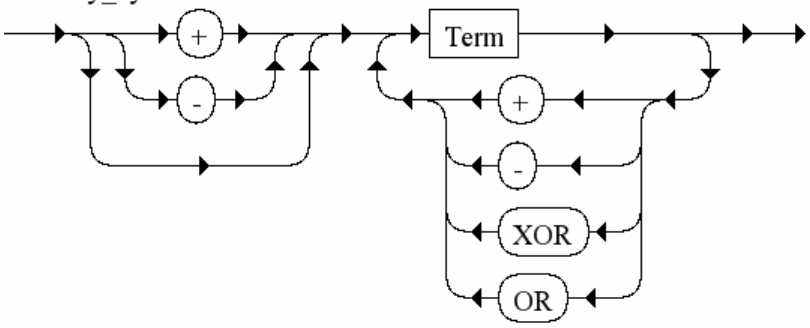
Prikaz



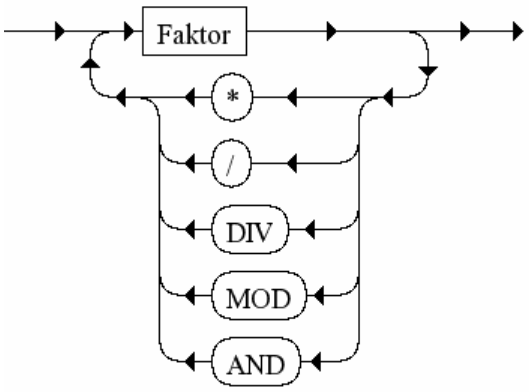
Vyraz



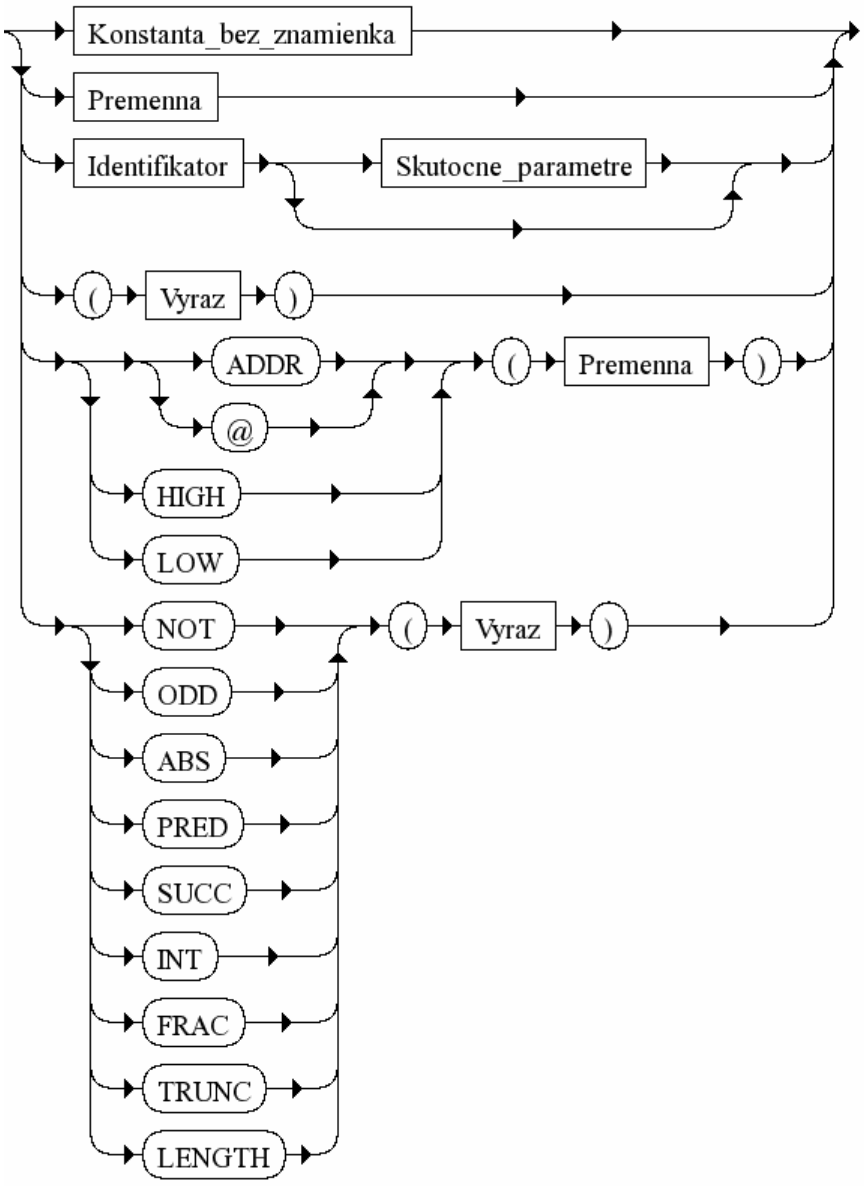
Aditivny_vyraz



Term



Faktor


Konstanta_bez_znamienka
Premenna
Identifikator
Skutocne_parametre
(
Vyraz
)
ADDR
@
(
Premenna
)
HIGH
LOW
NOT
(
Vyraz
)
ODD
ABS
PRED
SUCC
INT
FRAC
TRUNC
LENGTH