VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu IFJ a IAL

Implementace překladače imperativního jazyka IFJ17

Tým 129, varianta I.

6. prosince 2017

Seznam autorů:

Michal Tichý, xtichy26, 25 %

Michal Martinů, xmarti78, 25 %

Gabriel Mastný, xmastn02, 25 %

Ondřej Deingruber, xdeing00, 25 %

Obsah

1 Úvod

Tato dokumentace popisuje vývoj a implementaci překladače imperativního jazyka IFJ17. Dokumentace je rozdělena na části, které popisují celý vývoj překladače a jednotlivé prvky programového řešení.

Dále jsou zde přidány přílohy popisující strukturu konečného automatu a LL-gramatiky.

2 Zadání projektu

Cílem bylo vytvořit překladač imperativního jazyka IFJ17, který je podmnožinou open-source jazyku FreeBASIC. IFJ17 není case sensitive jazyk. Samotný překladač vygeneruje kód v jazyce IFJcode17, a pokud vše proběhlo bez chyby překladač vrátí návratovou hodnotu 0. V opačném případě je navrácen kód příslušné chyby.

Překladač je složen z následujících částí:

1. Lexikální analyzátor (scanner)
2. Tabulky symbolů
3. Syntaktický a sémantický analyzátor (parser)
4. Generátor kódu

Pro řešení jsme zvolili variantu zadání I., kde se pro řešení použil binární vyhledávácí strom.

3 Práce v týmu

3.1 Rozvržení práce

Po zveřejnění projektu byla v týmu svolána porada. Určilo se, kdo jakou práci vykoná. Důležité bylo také zvolit prostředky pro komunikaci v týmu, pro verzovací nástroj pro vývoj softwaru a v neposlední řadě vývojové prostředí.

U vývoje jednotlivých částí programu nebyly zvoleny přesné časy na vypracování. Spoléhalo se na spolehlivý osobní přístup každého člena týmu. Vedoucí přesto pravidelně kontroloval ostatní členy v průběhu řešení a případně dorozděloval úkoly.

3.2 Verzovací nástroj pro vývoj softwaru

Jedním ze základních pilířů skupinové práce mezi členy se stal verzovací nástroj Git a hostování repozitáře službou GitHub, se kterým již měli ostatní členové bohaté zkušenosti. Repozitář byl vytvořen jako soukromý, aby k němu neměly přístup ostatní týmy a nevznikl tak problém s únikem zdrojových kódů.

3.3 Komunikace v týmu

Písemná komunikace mezi členy probíhala převážně přes soukromou skupinu na stránkách Facebooku, kde byl každý člen aktivně k zastižení. Pro řešení náročnějších problémů jsme používali Skype. Díky těmto dvěma komunikačním kanálům se nám podařilo eliminovat potřebu osobních schůzek. Komunikace přes internet se tedy stala stěžejní pro celý tým.

3.4 Metodika vývoje software

Metodika vývoje nebyla předem přímo určena. Používali jsme vlastní testovací prostředí vytvořené v jazyce C. Každý člen si psal testy na vlastní část programu.

4 Implementace

V aktuální kapitole jsou popsány jednotlivé části software.

4.1 Lexikální analyzátor (Scanner)

Lexikální analyzátor postupně čte zdrojový kód a odstraňuje z něho přebytečné bílé znaky, komentáře a rozpoznává jednotlivé lexémy, z kterých vytváří odpovídající tokeny (klíčová slova, identifikátory, datové typy atd.). Podle typu je ukládána do tokenu jeho příslušná hodnota. Lexikální analyzátor je implementován jako konečný automat obsahující řídící switch s proměnnou state, do které se ukládá aktuální stav. Načtení tokenu je prováděno funkcí GetNextToken, která načte příslušný token do oboustranného lineárně vázáného seznamu. V případě lexikální chyby se program ukončí s návratovou hodnotou 1. Jak celý program, tak Lexikální analyzátor končí, pokud již není možné načíst další znak ze zdrojového souboru.

4.2 Syntaktický analyzátor

Pro syntaktickou analýzu jsme zvolili metodu rekurzivního sestupu.
Tato metoda si postupně bere tokeny ze scanneru a snaží se pomocí nich sestavit abstraktní syntakticky strom (AST). Sestavení stromu probíhá voláním vzájemně rekurzivních funkcí. Každá z těchto funkcí ověřuje jedno pravidlo gramatiky.

Pokud parser zjistí, že je splněno pravidlo, tak vrátí předcházející funkci uzel, který reprezentuje část programu, kterou se povedlo sestavit.

Pokud parser zjistí, že toto pravidlo splněno není, tak vrátí scanneru všechny tokeny, které si vyžádal, a do předcházející funkce vrátí NULL. Díky tomu předcházející funkce ví, že dané pravidlo nebylo splněno. Pak může zkusit jinou možnost, jak sestavit AST.

Například pokud bychom chtěli naparsovat tento program, tak parsování proběhlo voláním těchto funkcí:

4.2.1 Vstupní program

```
scope
        Dim a as string
        a=5
end scope
```

4.2.2 Volané funkce

- ProcessProgram() //pokusí se nalézt definice funkcí a hlavní scope
  - ProcessFunctionDefinitions() //nenalezne žádné funkce, pokračuje hledáním hlavního scope
  - ProcessScope() //nalezne hlavní scope
    - ProcessStatement() //pokusí se nalézt příkazy, které scope obsahuje
      - ProcessVariableDeclaration() //nalezne deklaraci proměnné
      - ProcessAssignment() //nalezne přiřazení
        - ProcessExpression() //nalezne výraz k přiřazení

Pokud se nepodaří AST sestavit, tak překladač vrací chybový kód 2.

5 Sémantický analyzátor

Sémantická analýza se provádí souběžně s analýzou syntaktickou.
Pokud je při zpracování uzlu možnost odhalení sémantické chyby, tak se provede příslušná kontrola.

Příkladem uzlu, kde je třeba tuto kontrolu provést, je přiřazení do proměnné. V tomto případě je třeba zkontrolovat, že datový typ proměnné a datový typ přiřazovaného výrazu jsou kompatibilní. Kontrolu datových typů ve většině případů řešíme pomocí funkce GetResultType, která na základě poskytnutých typů a operátorů rozhodne o datovém typu výsledku a také detekuje sémantický problém, pokud požadovaná operace není možná pro dané typy operandů.

Sémantická kontrola probíhá také při deklaraci proměnných. V tomto případě je nutné zkontrolovat, jestli tabulka symbolů na dané úrovni již neobsahuje proměnnou se shodným názvem.

Náročnější sémantická kontrola je potřeba při definici a deklaraci funkcí. Problém spočívá v nutnosti kontroly shodnosti deklarace a definice funkce (shodnosti v parametrech a návratovém typu). Deklarace nesmí následovat po definici, ale opačný případ je validní.

6 Tabulka symbolů

Implementace tabulky symbolů je určena zadáním. Implementace proběhla pomocí vyhledávacího binárního stromu. Binární strom je nevyvážený. To znamená, že výška levého a pravého podstromu se může lišit o více než o jeden prvek. Tabulka symbolů je rozdělena na tabulku neznámých a tabulku funkcí. Do tabulky neznámých se ukládají názvy proměnných. Každý prvek obsahuje jméno proměnné, které slouží jako unikátní vyhledávací prvek, ukazatel na levý prvek a ukazatel na pravý prvek. Pro každý scope je vytvořena nová tabulka symbolů. Prvek tabulky obsahuje jako unikátní klíč název funkce, ukazatel na levý prvek, ukazatel na pravý prvek, seznam parametrů, jejich datové typy a datový typ návratové hodnoty. Tabulka funkcí existuje pouze jednou.

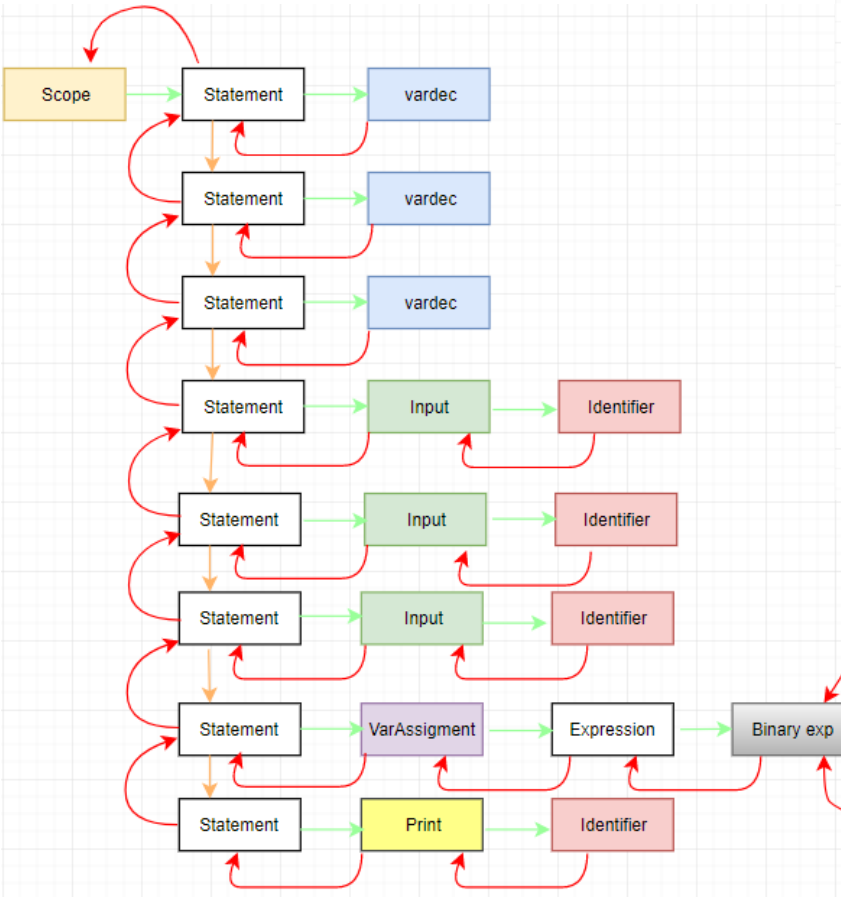
7 Generátor kódu

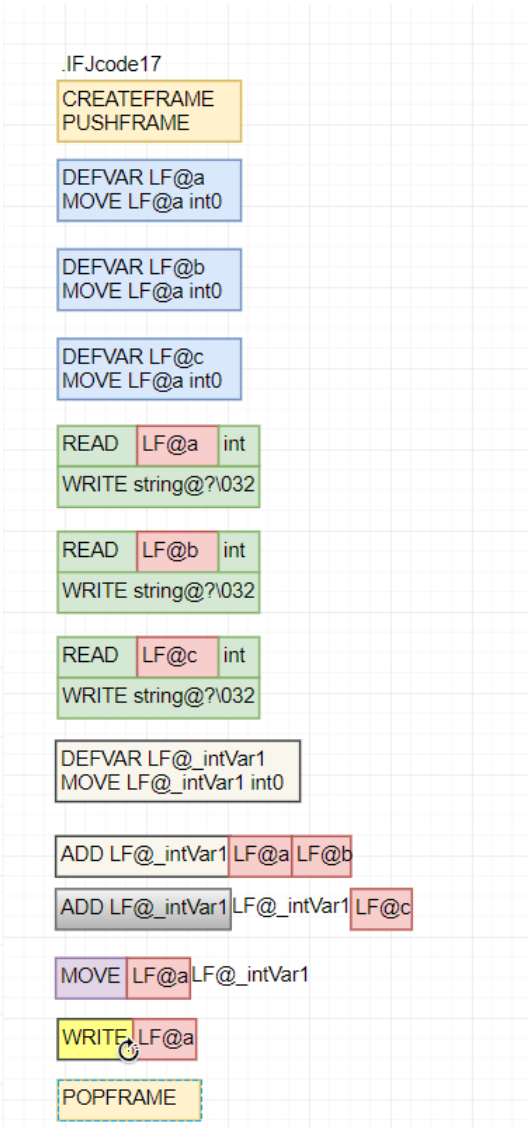
Srdce celého generátoru je funkce Recognize, která na základě abstraktního syntaktického stromu poskytnutého syntaktickým analyzátorem, rekursivně prochází jednotlivé uzly a podle jejich typu a dalších vlastností vytváří tříadresný kód, který je určen pro následující interpret.

Příklad jednoduché sčítačky:
jednotlivé zanořování je vyobrazeno šipkami s
následující prioritou: zelená nejvyšší - červená nejnižší

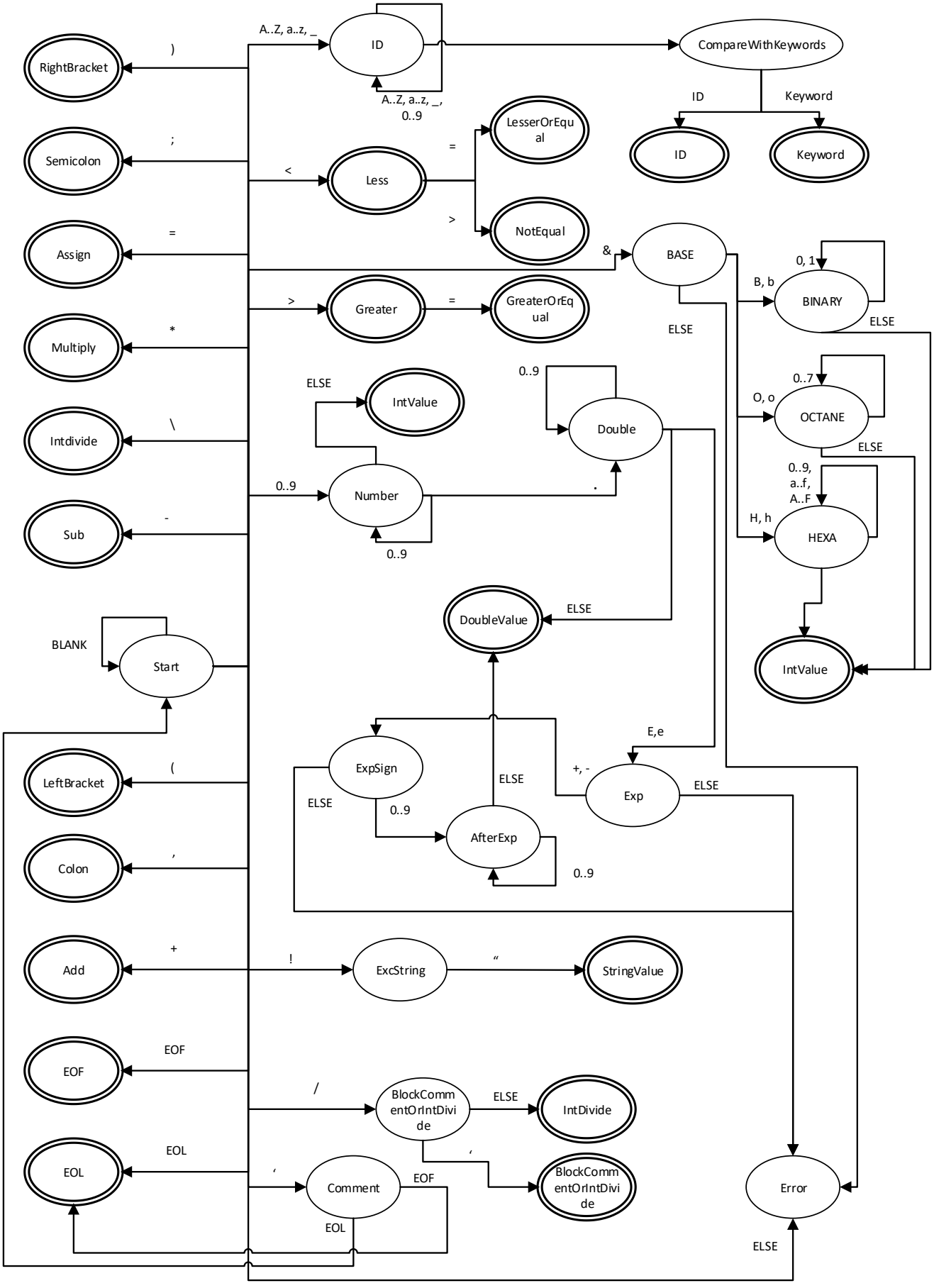
jazyk free Basic:

```
scope
dim a as integer
dim b as integer
dim c as integer
input a
input b
input c
a = a + b + c
print a;
end scope
```





8 Konečný automat



9 Pravidla LL-Gramatiky

- Program
  - FunctionDeclaration* statement_separator FunctionDefinition* statement_separator scope_statement
- FunctionDefinition
  - Function Identifier(Parameters) As Scalar Eol Statement* return Expression End Function
- FunctionDeclaration
  - Declare Function Identifier(Parameters) As Scalar
- Statement
  - statement_separator (Variable_declaration| procedure_call| compound_statement | quirk_statement | Assignment)? (statement_separator Statement)*
- statement_separator
  - Eol+
- procedure_call
  - Identifier(procedure_parameter_list)
- compound_statement
  - scope_statement
  - If_statement
  - for_statement
  - do_statement
  - While_statement
- scope_statement
  - scope Statement end scope
- If_statement
  - if Expression then Statement else Statement end if
- elseif_block
  - elseif Expression then Statement
- While_statement
  - Do While EXPRESSION STATEMENT LOOP
- Assignment
  - Identifier = Expression
- Variable_declaration
  - Dim Identifier As Scalar
  - Dim Identifier As Scalar = Expression
- Scalar
  - Integer
  - Double
  - String
- Parameters
  - Parameter(, Parameter)*

- Parameter
    - Identifier as Scalar
- procedure_parameter_list
    - procedure_parameter(, procedure_parameter)*
- procedure_parameter
    - Expression
- Expression
    - Logical_or_expression
    - String
- Logical_or_expression
    - logical_and_expression (or logical_and_expression)*
- relational_expression
    - add_expression ((=|>|<|<>|<=|>=) add_expression*
- add_expression
    - integer_division_expression (+ | -) iinteger_division_expression)*
- integer_division_expression
    - multiplication_expression (\ multiplication_expression)*
- multiplication_expression
    - prefix_expression ((* | /) prefix_expression)*
- prefix_expression
    - (-|+) prefix_expression
    - not relational_expression
- highest_precedence_expression
    - parenthesised_expression
    - Atom
- parenthesised_expression
    - (Expression)
- Atom
    - procedure_call
    - quirk_function
    - Identifier
    - Literal
- quirk_function
    - quirk_function_name procedure_parameter_list
- quirk_function_name
    - Input
    - Print
- quirk_statement
    - print_statement
    - Input_statement
- print_statement
    - Print expression(;expression)*
- Input_statement