Dokumentace k projektu z předmětů IFJ/IAL

Implementace překladače imperativního jazyka IFJ17

6.12.2017

Tým 025, varianta I

Naňo Andrej (vedúci týmu)	xnanoa00	25%
Marko Peter	xmarko15	25%
Švanda Jan	xsvand06	25%
Mechl Stanislav	xmechl00	25%

Obsah

Úvod

V tomto dokumentu popisujeme návrh a postup implementace překladače imperativního jazyka IFJ17. Začneme rozdělením práce a později popíšeme implementované metody.

Rozdělení práce mezi členy týmu

Andrej Naňo (xnanoa00)

- Návrh LL(1) gramatiky
- Vytvoření základní struktury parseru a implementování funkci pro jednotlivé neterminály

Peter Marko (xmarko15)

- Tabulka symbolů
- Syntaktický analyzátor pro výrazy

Ján Švanda (xsvand06)

- Lexikální analyzátor
- Testy

Stanislav Mechl (xmechl00)

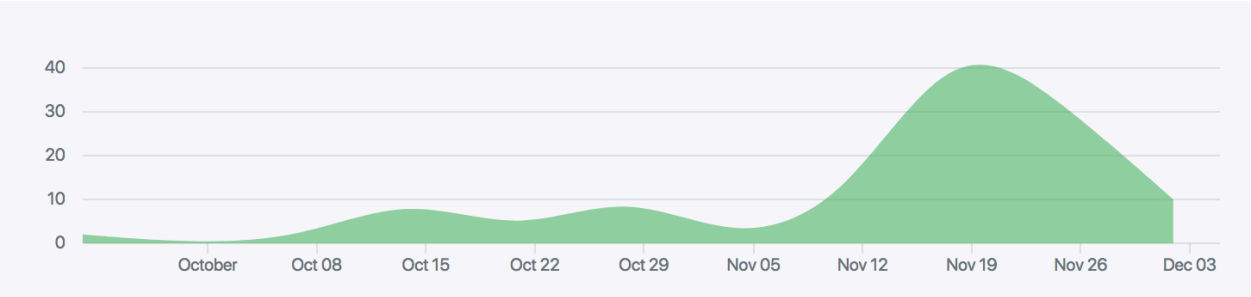
- Generování výstupního kódu
- Implementace vestavěných funkcí

Práce v týmu

Práci na projektu jsme začali už během října, od kdy jsme se pravidelně osobně setkávali jedenkrát týdně. Na lepší rozdělení a synchronizaci práce jsme využívali nástroj Git s repozitářem uloženým na serverech GitHub.

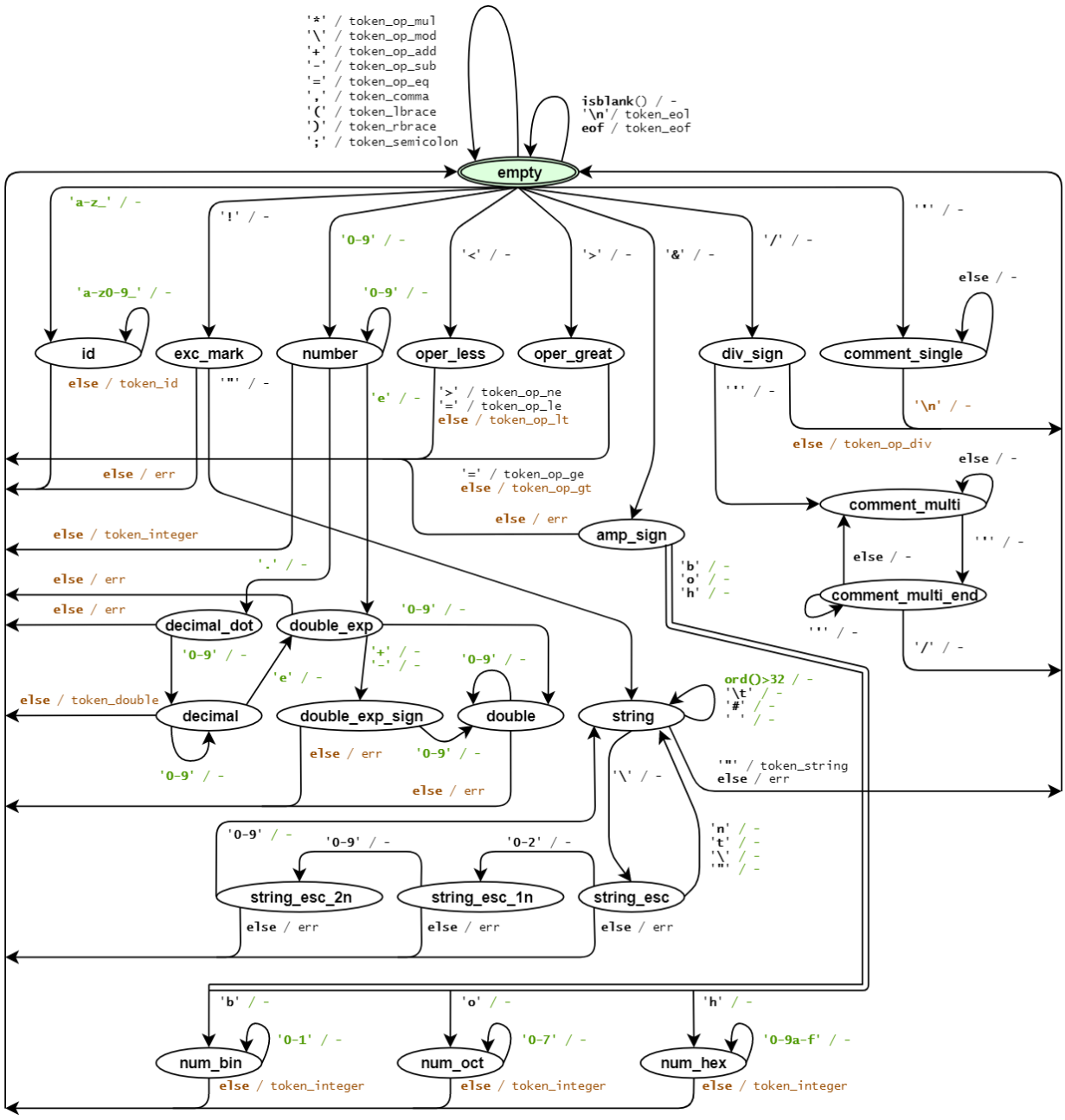
První setkání byly čistě teoretické a sloužily k zorientování se v zadání projektu. Úkoly jsme si rozdělili podle intuice, protože jsme v té době ještě nevěděli, jak by měl projekt fungovat. Nejvíce náročnou vlastností tohoto projektu bylo, že jsme se o způsobu řešení dané části projektu dozvěděli až v pozdějších týdnech semestru a nebyli jsme schopni na projektu dělat většinu času paralelně. Z toho důvodu se intenzivní práce na projektu objevila až zhruba 2 týdny před prvním zkušebním odevzdáním.

Znázornění množství práce vynaložené na projektu během semestru :



Lexikální analýza

Konečný stavový automat pro lexikální analýzu byl implementován podle následujícího diagramu. Jedná se o Mealyho automat, jehož vstupem je čtený znak ze zdrojového souboru a výstupem je token odpovídající poslupnosti vstupních znaků.



Poznámky k diagramu

Znázorněný konečný automat je modelem implementovaného automatu. Implementovaný automat se mírně liší v chování ohlašování chyb, kde může jeho činnost být přerušena násilně mimo koncový stav. Zelené přechody značí změnu hodnoty výstupního datového typu. Červené přechody označují návrat čteného znaku do zdrojového souboru, aby jej bylo možné znovu číst a znak nebyl vynechán.

Cíl lexikální analýzy

Cílem lexikálního analyzátoru, neboli scanneru, je převést posloupnost znaků (lexémů) ze zdrojového souboru do patřičné posloupnosti tokenů. Tokeny reprezentují různá klíčová slova, nebo číselné či řetězcové literály ze zdrojového souboru. Zároveň kontroluje správnost zápisu lexémů a odstraňuje bílé znaky a komentáře.

Alokace paměti

Při implementaci bylo nutné řešit několik problémů, které se hlavně týkaly oznamování vyskytnutých chyb a posléze správné uvolňování paměti. Problém s uvolňováním paměti jsme se rozhodli přesunout z větší části do oblasti parseru, tak aby scanner mohl bez problémů předat správný token nebo oznámit chybu.

V původní verzi probíhala alokace tokenů přímo v scanneru, tento způsob se později vzhledem k uvolňování paměti a časté akolaci a dealokaci tokenů projevil jako nepříliš vhodný. Později jsme tedy přešli k alokaci v parseru. Token je tedy předáván pouze ukazatelem a je tak předejito problému s jeho alokací.

Je ovšem nutné alokovat paměť pro string literály a identifikátory, k tomu slouží skupina funkcí ve scanneru, která se stará o přidávání znaků do řetězce a o správnou alokaci paměti. Paměť je alokována po několika bytech, tak aby nebyly alokace příliš časté.

Oznámení vyskytnuté chyby

Dále bylo potřeba vyřešit oznámení vyskytnutých chyb, například při čtení double. Zvolili jsme způsob, při kterém měníme stav globální chyby a zachovává se první oznámená. Pokud tedy dojde k chybě, scanner změní globální stav a vrátí token token_eof který značí ukončení souboru. Takto jsme se rozhodli, protože nebyl další důvod pokračovat v překladu, ze zadání vyplynulo oznámit výskyt první chyby.

Syntaktická a sémantická analýza

Syntaktický analyzátor jsme implementovali pomocí rekurzivního sestupu podle LL(1) gramatiky.

Pro každý neterminál v gramatice byla vytvořena void funkce.
Syntaktický analyzátor měl za úkol:

- Zkontrolovat syntaxi příchozích tokenů ze scanneru
- Spustit další funkce neterminálů v závislosti na právě čteném tokenu
- Udělat sémantické kontroly v kontextu, který to vyžaduje
- Pokud proběhne vše správně, tak se vygeneruje nová instrukce a přidá se do seznamu instrukcí
- V případě chyby se nastaví globální proměnná signalizující chybu, při ukončení parseru se chyba vypíše a instrukce se na výstup nevypíše

Vyhodnocování výrazů zabezpečuje parser pro výrazy, který se řídí precedenční tabulkou.

LL – gramatika

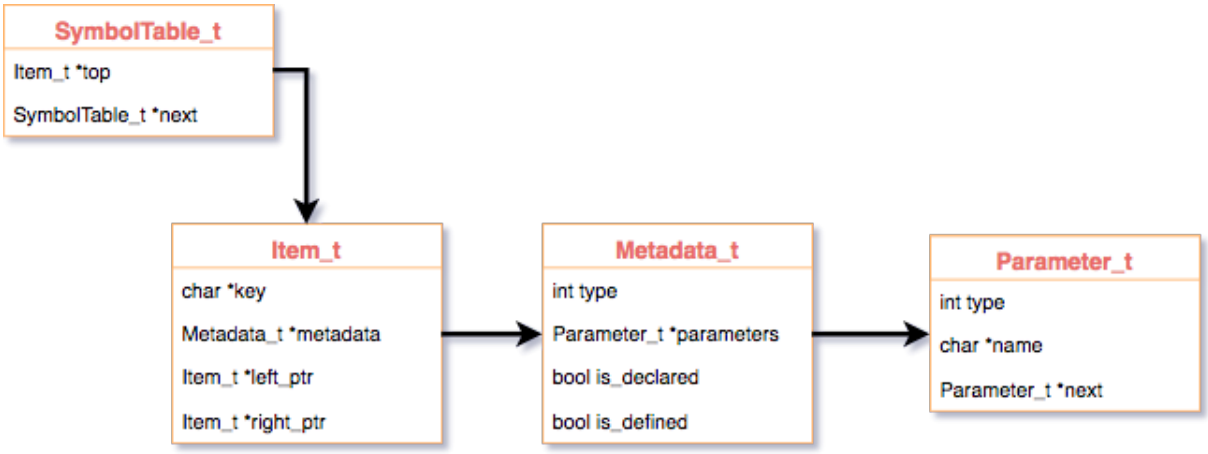
NONTERMINALS

```
  <Program>        →   <Head> <Scope> token_eof

     <Head>        →   ε
     <Head>        →   <FunctionDecl> <Head>
     <Head>        →   <FunctionDef> <Head>

    <Scope>        →   token_scope <CompoundStmt> token_endscope

<FunctionDecl>     →   token_declare <Function>
 <FunctionDef>     →   <Function> token_eol <CompoundStmt> token_end token_function
  <Function>       →   token_function token_identifier token_lbrace <ParameterList> token_rbrace token_as token_datatype

  <ParamList>      →   ε
  <ParamList>      →   <Param> <NextParam>
  <NextParam>      →   token_comma <Param> <NextParam>
  <NextParam>      →   ε
    <Param>        →   token_identifier token_as token_datatype

<CompoundStmt>     →   ε
<CompoundStmt>     →   <VarDec> <CompoundStmt>
<CompoundStmt>     →   <VarDef> <CompoundStmt>
<CompoundStmt>     →   <AssignStmt> <CompoundStmt>
<CompoundStmt>     →   <ReturnStmt> <CompoundStmt>
<CompoundStmt>     →   <WhileStmt> <CompoundStmt>
<CompoundStmt>     →   <CallStmt> <CompoundStmt>
<CompoundStmt>     →   <IfStmt> <CompoundStmt>

    <VarDec>       →   token_dim token_identifier token_as token_datatype
    <VarDef>       →   <VarDec> token_equals <Expr>
 <AssignStmt>      →   token_identifier token_equals <Expr>
 <ReturnStmt>      →   token_return <Expr>
 <WhileStmt>       →   token_do token_while <Expr> token_eol <CompoundStmt> token_loop
   <IfStmt>        →   token_if <Expr> token_then token_eol <CompoundStmt> token_else token_eol <CompoundStmt> token_end token_if
 <CallStmt>        →   token_identifier token_equals token_identifier token_lbrace <TermList> token_rbrace token_eol
 <InputStmt>       →   token_input token_identifier
 <PrintStmt>       →   token_print <ExprList>

  <ExprList>       →   <Expr> token_semicolon <NextExpr>
  <NextExpr>       →   <ExprList>
  <NextExpr>       →   ε

  <TermList>       →   ε
  <TermList>       →   <Term> <NextTerm>
  <NextTerm>       →   token_comma <TermList>
  <NextTerm>       →   ε

    <Term>         →   ε
    <Term>         →   token_integer
    <Term>         →   token_double
    <Term>         →   token_string
    <Term>         →   token_identifier
```

Precedenční tabulka vyhodnocení výrazů

	+	-	*	/	\	>	<	>=	<=	<>	=	(	)	ID	$
+	>	>	<	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
\	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	>	>	>	>	>	>	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>				>
ID	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	

Tabulka symbolů



Implementace tabulky symbolů

Tabulky symbolů jsou implementovány jako seznam binárních vyhledávacích stromů, přičemž každá položka v seznamu je vlastně tabulka obsahující proměnné deklarované v konkrétním scope, při vyhledávání se prochází nejprve prvním stromem v seznamu a pak dalšími, tímto způsobem je zajištěna priorita proměnných uvnitř scope.
Implementace tabulky pro proměnné a funkce je stejná a proto je tomu přizpůsobena i datová struktura

Struktura položky v tabulce symbolů:

```
typedef struct Metadata
{
        int type;                       datový typ položky
        Parameter_t *parameters;        seznam parametrů funkce
        bool is_defined;                položka byla definována
        bool is_declared;               položka byla deklarována
} Metadata_t;
```

Generování kódu

Jednotlivé instrukce jsou generovány jako vnitřní tříadresní kód a ukládány do jednosměrně vázaného seznamu. Po úspěšném provedení syntaktické a sémantické analýzy je na výstup vypsána nejdříve hlavička programu, následně použité vestavěné funkce a poté zbývající instrukce.

Závěr

Nejtěžší částí projektu bylo pochopit zadání a spravedlivě rozdělit úkoly mezi jednotlivé členy týmu tak, aby každý dostal úkol který mu nejvíce sedí. Rozhodli jsme se že se budeme každý týden setkávat a diskutovat o nových problémech, tento systém práce byl podle mého názoru úspěšný a v našem týmu byla proto dobrá spolupráce. Po dlouhých týdnech tvrdé práce na překladači a mnoha probdělých nocích práce, testování a neustálého zlepšování kódu, se nám konečně podařilo vytvořit relativně funkční překladač. Dali jsme do toho absolutně všechno a podařilo se nám vytvořit to co se nám na začátku zdálo téměř neproveditelné. Naučili jsme se mnoho nových věcí a zjistili jsme jak fungují překladače, které každodenně používáme.